XVT-Power++™

A C++ Application Framework from the #1 Company in Multi-Platform Development Tools.

XVT

XVT-Power++

Volume 1

XVT-POWER++ INSTALLATION

XVT - THE PORTABLE GUI DEVELOPMENT SOLUTION

Version 1.1

Published By

XVT Software Inc.
Box 18750
Boulder, CO 80308
(303) 443-4223
(303) 443-0969 (FAX)

Credits

Architecture and Design:
Michael Sher

Engineers:
Michael Sher, Ted Kremer, Carol Meier, InfoStructure Inc.

Documentation:
Noreen Garrison, Paul Tepley, Cindi Fisher, Suanna Jo Schamper

Product Team:
Dave Locke, Michael Sher, Jonathan Auerbach, Catherine Connor, Peggy Reed, Connie Leserman

Printing History

First Printing November, 1993 XVT-Power++ Version 1.1

This manual is printed on recycled paper by Continental Graphics, Broomfield, Colorado.

XVT-POWER++

CONTENTS

XVT-POWER++

INSTALLATION

I.1. Introduction

This appendix gives instructions for installing and starting XVT-Power++ Version 1.1 on three different platforms:

- UNIX Motif/OPEN LOOK
- Windows
- Macintosh

You need to read only the section that is relevant to you, as well as the release notes that are included with your distribution disk.

I.2. UNIX Motif/OPEN LOOK

This section tells you how to install and start XVT-Power++ under UNIX Motif/OPEN LOOK. The installation breaks down into the following general tasks:

- Installing the program
- Setting up the environment
- Completing the XVT-Power++ 1.1 release requirements
- Compiling the XVT-Power++ library
- Compiling the XVT-Power++ examples
- Using the XVT-Power++ shell project

I.2.1. Installing the Program

▼ To install XVT-Power++:

1. Obtain root privileges on the target machine. Insert the distribution tape into the tape drive and enter the following command (assuming /dev/rst0 corresponds to your tape drive):

```
tar -xf /dev/rst0
```

This step extracts a tar file named **xvtpwr.tar**.

2. Move or `ftp` this file to the directory where you wish to install XVT-Power++. For example, to install XVT-Power++ in **/usr**, enter the following commands:

```
mv xvtpwr.tar /usr
cd /usr
```

3. Finish untarring the file with the following commands:

```
tar -xpf ./xvtpwr.tar
rm xvtpwr.tar
```

The following directories are created:

.../xvtpwr/util	XVT-Power++ UNIX utilities
.../xvtpwr/doc	XVT-Power++ release documentation
.../xvtpwr/src	XVT-Power++ source files
.../xvtpwr/include	XVT-Power++ header files
.../xvtpwr/lib	XVT-Power++ library
.../xvtpwr/demo	Demo applications
.../xvtpwr/demo/Ask	Example of using CWindow as a dialog
.../xvtpwr/demo/Drag	Drag and drop example
.../xvtpwr/demo/Draw	Tutorial drawing program
.../xvtpwr/demo/Edit	Tutorial text editor
.../xvtpwr/demo/HardHat	Prototype of a graphical builder
.../xvtpwr/demo/Hello	Tutorial "Hello World" program
.../xvtpwr/demo/Shell	XVT-Power++'s "Shell" application
.../xvtpwr/demo/SpreadSheet	Spreadsheet example

I.2.2. Setting Up the Environment

Once XVT-Power++ is installed, define the following environment variables:

`PWRDIR`
: the pathname of the XVT-Power++ directory (for example: **/usr/Pwr**)

`XVTDIR`
: the pathname of the XVT directory containing the XVT

Portability Toolkit and Motif subdirectories (for example: `/usr/xvtxm`)

You may also want to add the path to **.../Pwr/util** to your environment `PATH` variable.

I.2.3. Completing XVT-Power++ Version 1.1 Release Requirements

Following are the requirements for the Version 1.1 release of XVT-Power++:

XVT Portability Toolkit

You must have the current version of XVT/XM or XVT/XOL installed. Make sure the installation was successful by compiling and testing some of the example programs provided by the XVT Portabiltiy Toolkit.

Motif

A variety of C++ compilers under different platforms have been used to compile XVT-Power++ programs. Make sure your particular compiler is at least AT&T CFront 2.1 compliant.

Under some C++ compilers, you may need to do some tune-up work to compile and link with XVT. In some cases you will have to modify the **xvt_cc.h** file to enable compiling. Please call XVT if you need further assistance.

Other

In addition to the previous requirements, make sure you have read the XVT/XM installation requirements and that your system complies.

I.2.4. Compiling the XVT-Power++ Library

To build the XVT-Power++ library (**Pwr.a**), first change directories to **Pwr/src** and then execute make:

```
cd .../Pwr/src
make
```

You may need to fine tune the makefile provided in **src**. The makefile contains instructions on how to do this.

I.2.5. Compiling the XVT-Power++ Examples

XVT-Power++ provides several example applications inside the **Pwr/demo** directory. Each example subdirectory contains the source and makefiles needed to create the appropriate program. You

may need to modify the files in the manner described in Section I.2.4. In general, you can simply change to the example directory and execute `make` using the directory's name as the `make` target. For example, to create the HardHat demo you would enter:

```
cd .../Pwr/demo/HardHat
make HardHat
```

Several of the examples provided are described in detail in the tutorial (Chapter 3 of this *Guide*). You will benefit from reading at least the first three chapters of this manual (including the tutorial) before actually examining the code provided. Also, keep in mind that the classes and methods you encounter are fully described in the *XVT-Power++ Reference*.

I.2.6. Using the XVT-Power++ Shell Project

XVT-Power++ supplies a shell application in the **demo/shell** directory. This application consists of the basic example shell application framework classes: `CShellApp`, `CShellDoc`, and `CShellWin`. Information on each of these classes is provided in the *XVT-Power++ Reference*. The shell project provides a quick and easy way to get started. In most cases you should use these files when starting a new application rather than starting from scratch. In addition, you will need these files to work through the XVT-Power++ tutorial in Chapter 3.

Under the XVT-Power++ UNIX platform, you should use and reuse the shell project by using the `newdemo` utility provided in the **Pwr/util** directory. For example, to create a new XVT-Power++ application named *Test*, enter the following:

```
newdemo Test
cdTest
make
```

`newdemo` will obtain fresh copies of the Shell project files and rename them using `Test` as a base name. It also creates a makefile that will make the newly created `Test` project.

I.3. Microsoft Windows

This section tells you how to install and start XVT-Power++ under Windows. The installation breaks down into the following general tasks:

- Installing the program
- Setting up the environment

- Completing XVT-Power++ Version 1.1 release requirements
- Compiling the XVT-Power++ library
- Creating resource (**.rc** and **.res**) files
- Compiling the XVT-Power++ examples
- Using the XVT-Power++ shell project

I.3.1. Installing the Program

This XVT-Power++ release is distributed in a self-extracting archive named **xtvpwr.exe**.

▼ To install XVT-Power++:

Copy **xvtpwr.exe** and execute it. You can do this either from the Windows File Manager or directly from DOS as in the following example:

```
C> copy a:xvtpwer.exe
C> xvtpwr.exe
```

The following directories are created:

xvtpwr\src	XVT-Power++ source files
xvtpwr\include	XVT-Power++ include files
xvtpwr\lib	XVT-Power++ libraries
xvtpwr\demo	XVT-Power++ demos
xvtpwr\demo\ask	Example of using CWindow as a dialog window
xvtpwr\demo\drag	Drag and drop example
xvtpwr\demo\draw	Tutorial drawing program
xvtpwr\demo\edit	Tutorial text editor program
xvtpwr\demo\hardhat	Prototype of an XVT-Power++ graphical builder
xvtpwr\demo\hello	Tutorial "hello world" program
xvtpwr\demo\shell	XVT-Power++'s "Shell" application
xvtpwr\demo\sprdsht	Spreadsheet example

I.3.2. Setting Up the Environment

Set up the environment variables `LIB`, `INCLUDE`, and `XVTDIR` by editing your **AUTOEXEC.BAT** file. If these variables are already defined, add the XVT-Power++ information as illustrated in the following example:

```
SET INCLUDE=C:\COMPILER\INCLUDE;C:\xvtpwr\INCLUDE
SET LIB=C:\COMPILER\INCLUDE;C:\xvtpwr\LIB
SET XVTDIR=c:\XVT
```

If the variables are not defined, define them as follows:

```
SET INCLUDE=C:\xvtpwr\INCLUDE
SET LIB=C:\xvtpwr\LIB
SET XVTDIR=c:\XVT
```

I.3.3. Completing XVT-Power++ Version 1.1 Release Requirements

Following are the requirements for the Version 1.1 release of XVT-Power++:

XVT Portability Toolkit

You must have XVT/Win installed. Make sure the installation was successful by compiling and testing some of the example programs provided by the XVT Portability Toolkit.

Compilers

Following is a list of C++ compilers for Windows. In addition to following the XVT-Power++ requirements and suggestions for each compiler, be sure to follow all XVT Portability Toolkit guidelines and suggestions.

Borland C++ 3.1

This release fully supports this version of Borland. Example project files are included. These project and makefiles assume that Borland can be located in the root directory of the **c:\o** drive. You can achieve this by first modifying your **CONFIG.SYS** file to include the following command and then rebooting your machine:

```
LASTDRIVE=O
```

Assuming your compiler is installed in **C:\BORLANDC**, map the drive by typing in the following command at the DOS prompt or by including it in your **AUTOEXEC.BAT** file:

```
SUBST O: C:\BORLANDC
```

Microsoftt Visual C++

This XVT-Power++ release fully supports this version of Microsoft's C++ compiler. Example makefiles are included.

Memory Models

The current Version 1.1 release of XVT-Power++ supports only the large memory model. Make sure you follow the compilation instructions by compiling under the correct

memory model and linking to the appropriate XVT Portability Toolkit libraries.

Note: In addition to the previous requirements, make sure you have read the XVT/WIN installation requirements and that your system complies.

I.3.4. Compiling the XVT-Power++ Library

A ready-to-use XVT-Power++ library is already provided with your installation. Inside the **xvtpwr\lib** directory, you will find the following:

lbPwr1.lib	Large memory model XVT-Power++ libraries for Borland
lbPwr2.lib	
lmPwr.lib	Large memory model XVT-Power++ library for Microsoft
lmPwr2.lib	

These libraries do not contain debugging or browsing information, but you can easily rebuild them with the information below.

Note: If compiling from scratch, copy the libraries to **xvtpwr/lib** first.

Recompiling Under the Borland Environment:

You can rebuild the XVT-Power++ library using Borland and the project file provided in your installation. Simply launch Borland and open the **lbPwr1.prj** and the **lbPwr2.prj** project in the **Pwr\lib** directory. Before invoking MAKE (from the Compile menu), you may need to modify the project to work under your own environment. Particularly, make sure the following settings are defined correctly:

- The **INCLUDE** directory settings contain the correct paths to Borland's and XVT Portability Toolkit's **include** directories.
- The **LIB** directory settings contain the correct paths to Borland's and XVT Portability Toolkit's library directories.

Recompiling under Microsoft Visual C++:

You can rebuild the XVT-Power++ library by using the makefiles provided in the **xvtpwr\src** directories, named

lmPwr1.mak and **lmPwr2.mak**. You can use this makefile under the Microsoft workbench.

I.3.5. Creating Resource (.rc and .res) Files

Creation of resource files must be done outside the compiler environment. You must modify the XVT Portability Toolkit's **MAKERES.BAT** ultility, adding `-d DOS` to the CURL compile line.

To use **MAKERES.BAT**, enter the URL name as follows:

```
makeres ask
```

I.3.6. Compiling the Examples

XVT-Power++ provides several example applications inside the **xvtpwr\demo** directory. Each example subdirectory contains the source, Borland project file (**.prj**), and Microsoft Visual C++ makefiles (**.mak**) needed to create the appropriate program. You may need to modify the project files as described in Section I.3.4. In addition, make sure that the XVT Portability Toolkit libraries and files used by each project are located in the correct directories. Finally, follow the instructions on creating resource files in Section I.3.5..

Several of the examples provided are described in detail in the tutorial (Chapter 3 of this *Guide*). You will benefit from reading at least the first three chapters of this manual (including the tutorial) before actually examining the code provided. Also, keep in mind that the classes and methods you encounter are fully described in the *XVT-Power++ Reference*.

I.3.7. Using the XVT-Power++ Shell Project

XVT-Power++ supplies a shell application in the **demo\shell** directory. This application consists of the basic example shell application framework classes: `CShellApp`, `CShellDoc`, and `CShellWin`. Information on each of these classes is provided in the *XVT-Power++ Reference*. The shell project provides a quick and easy way to get started. In most cases, you should use these files when starting a new application rather than starting form scratch. In addition, you will need these files to work through the XVT-Power++ tutorial.

Note: When you have completed building the shell project, rename **shell.exe** to **shell1.exe**.

Under the XVT-Power++ Microsoft Windows platform, you should use and reuse the shell project by using the File Manager to make fresh duplicate copies of the entire **xvtpwr\demo\shell** directory. A future XVT-Power++ release will provide an automated method of creating a new shell application with a user-defined name (as is already done under UNIX and on the Macintosh).

I.4. Macintosh

This section tells you how to install XVT-Power++ on the Macintosh. The installation breaks down into the following phases:

- Installation
- Setting up the environment
- Meeting XVT-Power++ Version 1.1 release requirements
- Compiling the XVT-Power++ library
- Compiling the XVT-Power++ examples
- Using the XVT-Power++ shell project
- Learning about known bugs

I.4.1. Installation

▼ To install XVT-Power++:

1. Create a new folder, named PWR, inside your XVT folder.
1. Insert the distribution disk into the disk drive and double click on the "Power++.sea" icon.
2. When you are prompted for a location in which to extract the XVT-Power++ files, specify the new PWR folder.

The following folders are created:

xvtpwr:src	XVT-Power++ library source files
xvtpwr:include	XVT-Power++ library header files
xvtpwr:lib	XVT-Power++ library
xvtpwr:util	XVT-Power++ MPW utilities
xvtpwr:doc	XVT-Power++ release documentation
xvtpwr:demo	XVT-Power++ example applications

xvtpwr:demo:Ask	Example of CWindow used as a dialog window
xvtpwr:demo:Draw	Tutorial example drawing program
xvtpwr:demo:Edit	Tutorial example text editor
xvtpwr:demo:HardHat	Prototype of an interactive GUI builder
xvtpwr:demo:Hello	Tutorial "Hello World"
xvtpwr:demo:IconDrag	Drag and drop example
xvtpwr:demo:Shell	XVT-Power++'s Shell application
xvtpwr:demo:Spreadsheet	Spreadsheet example

I.4.2. Setting Up the Environment

The startup file that you created when you installed the XVT/Mac Portability Toolkit (**UserStartup•xvt**) contains the necessary XVT-Power++ environment definitions. Add the following:

```
Set XVTPWR {XVT}:pwr        # Folder containing XVT-Power++ files
Export XVTPWR
```

Modify these variable settings and make sure they are put into the MPW folder.

I.4.3. Completing XVT-Power++ Version 1.1 Release Requirements

The following are requirements of the current Version 1.1 release of XVT-Power++:

XVT Portability Toolkit

You must have XVT 3.02 or later installed. Make sure the installation was successful by compiling and testing some of the example programs provided by the XVT Portability Toolkit.

Compiler

Below is a list of C++ compilers for the Macintosh. In addition to following the XVT-Power++ requirements and suggestions for each compiler, make sure you follow all XVT Portability Toolkit guidelines and suggestions.

MPW C++

This Version 1.1 release fully supports the C++ version of MPW. Example makefiles and utility scripts are included.

Other

In addition to the previous requirements, make sure you have read the XVT/MAC installation requirements and that your system complies.

I.4.4. Compiling the XVT-Power++ Library

A ready-to-use XVT-Power++ library is already provided with your installation. The library is **Pwr.lib.o** and is located in the **Pwr:lib** directory.

You can rebuild the XVT-Power++ library using MPW and the makefile provided in your installation. Simply launch MPW, set **{XVTPwr}source** as the current directory, and build **PWR**.

I.4.5. Compiling the Examples

XVT-Power++ provides several example applications inside the **Pwr:demo** folder. Each example subdirectory contains the source and the MPW makefile needed to create the appropriate program. To build any example, change directories to the appropriate folder and invoke `Build`. The name of the build target is the same as the name of the example program's folder.

Several of the examples provided are described in detail in the tutorial (Chapter 3 of this*Guide*). You will benefit from reading at least the first three chapters of this manual (including the tutorial) before actually examining the code provided. Also, keep in mind that the classes and methods you encounter are fully described in the *XVT-Power++ Reference*.

I.4.6. Using the XVT-Power++ Shell Project

XVT-Power++ supplies a shell application in the **demo:shell** directory. This application consists of the basic example shell application framework classes: `CShellApp`, `CShellDoc`, and `CShellWin`. Information on these classes is provided in the *XVT-Power++ Reference*. The shell project provides a quick and easy way to get started. In most cases you should use these files when starting a new application rather than starting from scratch. In addition, you will need these files to work through the XVT-Power++ tutorial.

Under the XVT-Power++ Macintosh platform, you should use and reuse the shell project by means of the `NewPwrProject` script

provided in the **Pwr:util** directory. For ease of use, you should add this script to your execution path.

For example, to create a new XVT-Power++ application named `Test`, enter the following line at the MPW worksheet:

```
NewPwrProject Test
```

`NewPwrProject` will obtain fresh copies of the Shell project files and rename them using `Test` as a base name. It also creates a makefile that will make the newly created `Test` application. For more information on using these shell classes, consult Chapter 3 of the *XVT-Power++Guide.*

XVT-POWER++ GUIDE

XVT - THE PORTABLE GUI DEVELOPMENT SOLUTION

Version 1.1

Published By

XVT Software Inc.
Box 18750
Boulder, CO 80308
(303) 443-4223
(303) 443-0969 (FAX)

Credits

Architecture and Design:
Michael Sher

Engineers:
Michael Sher, Ted Kremer, Carol Meier, InfoStructure Inc.

Documentation:
Noreen Garrison, Paul Tepley, Cindi Fisher, Suanna Jo Schamper

Product Team:
Dave Locke, Michael Sher, Jonathan Auerbach, Catherine Connor, Peggy Reed, Connie Leserman

Printing History

First Printing November, 1993 XVT-Power++ Version 1.1

This manual is printed on recycled paper by Continental Graphics, Broomfield, Colorado.

XVT-POWER++

CONTENTS

Preface

Introducing XVT-Power++

XVT-Power++ is a C++ application framwork that allows application programmers to produce extensive graphical user interfaces with relatively few lines of code. XVT-Power++ features an object-oriented design that incorporates an application framework and encapsulates the complex work associated with GUI programming into several hierarchies of object classes. Application designers can use these objects as building blocks to assemble into programs. Thus, XVT-Power++ dramatically reduces application development time and effort.

XVT-Power++ works with most major GUI platforms, including Presentation Manager™, Motif®, OPEN LOOK®, MS-Windows™, and Macintosh®. XVT-Power++ is "portable" in that it offers one body of code for all targeted platforms. In most cases, once you develop a program on one platform, you can take the program to another platform, recompile it, and run it without further changes.

XVT-Power++ achieves portability through the use of the XVT Portability Toolkit. Consisting of a set of C functions that communicate with the underlying graphical toolkits of several machines, the XVT Portability Toolkit provides a portable layer between the application program and the underlying graphical system.

Because XVT-Power++ predefines a number of GUI components, program development is easier and faster. Whether an application needs scrollers, list boxes, validating text fields, grids, user-sizeable objects, buttons, or other similar features, XVT-Power++ provides classes that can simply be assembled. On the other hand, XVT-Power++ does not confine the GUI programmer to predefined XVT-Power++ objects. Using C++ and the XVT Portability Toolkit,

you can extend and modify the entire class library by overriding methods and creating new classes.

The XVT-Power++ Documentation

The XVT-Power++ documentation consists of two main parts, the *XVT-Power++ Guide* (this manual) and the *XVT-Power++ Reference*. The object of the *Guide* is to describe XVT-Power++'s structure, survey the overall functionality that is available in related groups of classes, and explain how things work in XVT-Power++. In short, it gives you the total picture. When you want details on a particular class, consult its section in the alphabetical *XVT-Power++ Reference*.

What You Already Need to Know

Throughout the XVT-Power++ documentation, we assume that you have some basic knowledge of GUI features and programming, a working knowledge of C++, and access to the XVT Portability Toolkit documentation.

Conventions Used in This Manual

In this manual, the following typographic conventions indicate different types of information:

`code`
: This typestyle is used for code and code elements (names of functions, attributes, options, flags, and so on). It also is used for environment variables, commands, program names, and executable names.

filenames
: Bold type is used for filenames and directory names.

emphasis
: Italics are used for emphasis and the names of documents.

▼ This triangle symbol marks the beginning of a procedure having numbered steps. These symbols can help you quickly locate "how-to" information.

Note: An italic heading like this marks a standard kind of information: a Note, Caution, Example, Tip, or See Also (cross-reference).

Contents of This Manual

This manual is organized into the following sections and chapters:

1. **Overview**
   Surveys the roles assigned to different parts of the XVT-Power++ system, showing how the various components interact with one another. The emphasis is on the specific development tasks that the application developer must perform.

2. **Coding Conventions and Style Guidelines**
   Describes XVT-Power++'s file structure, naming conventions, and the mangling utility that ensures that the names of XVT-Power++ classes will not clash. It also gives the C++ guidelines that XVT-Power++ follows in various areas.

3. **XVT-Power++ Tutorial**
   Guides you step-by-step through the process of using XVT-Power++ to build three graphical user interface applications in increasing order of complexity: an interactive windowing program, a simple file editor program, and a drawing program.

4. **Applications**
   Describes the startup and shutdown mechanisms of XVT-Power++'s application class and surveys the tasks performed at the application level.

5. **Documents**
   Explains how data is propagated within the XVT-Power++ system and describes in detail the different tasks that are performed at the document level in XVT-Power++'s application framework: accessing data, building windows, managing data, and managing windows.

6. **Views and Subviews**
   Explains how views work in XVT-Power++ and discusses the tasks that views handle, the characteristics of views, some useful categories within the view object hierarchy, the XVT-Power++ coordinate system, use of the mouse, and how messages are propagated among certain kinds of views.

7. **Application Framework**
   Describes how XVT-Power++'s application, document, and view classes fit together to compose XVT-Power++'s

application framework. It explains how messages are propagated throughout this structure and focuses on aspects of the framework that are of concern to any XVT-Power++ application developer: setting up menus and handling menu commands, handling keyboard events, defining the look-and-feel of your application by setting display properties such as colors and fonts and drawing modes, and becoming familiar with the printing facility.

8. **Windows**
   Discusses the attributes a window can have, the possible types of windows, how they are constructed, and how you can derive your own window classes. It briefly considers two example classes of windows that are derived from the main window class, the shell window and the task window. The chapter concludes with a look the desktop, which manages the layout of windows on the screen.

9. **Shapes**
   Surveys the different kinds of shape objects available in XVT-Power++, considers the resources for drawing them, and offers guidelines for when you can most appropriately use them rather than directly calling the XVT Portability Toolkit's drawing functions.

10. **Icons**
    Describes the basic functionality of XVT-Power++ icons and then briefly looks at how two variant classes extend this functionality.

11. **Virtual Frames**
    Discusses the virtual frame class, which consists of a large virtual area and a smaller display area. It also discusses the classes that inherit from the virtual frame class, such as list boxes and scrollers.

12. **Wire Frames and Sketchpads**
    Considers the wire frame class that is used for sizing and moving view objects, as well as the sketchpad class, which uses wire frames to sketch shapes on a drawing area of the screen.

13. **Text and Text Editing**
    Discusses XVT-Power++'s two overall text facilities: XVT-Power++'s own static text drawing class and a set of native text editing classes that harness the text editing capabilities of the XVT Portability Toolkit.

14. **Native Views**
    Surveys all of XVT-Power++'s native view classes, which provide control objects such as scrollbars, buttons, list boxes, radio buttons, check boxes, and pop-down menus, all of which have the look-and-feel of the native window manager and provide some means for the user to interact with the application.

15. **Grids**
    Gives an overview of the functionality that is available through XVT-Power++'s three grid classes, a base class and two variant classes that allow you to create either a grid in which the cells are all the same size or a grid with variable-sized cells.

16. **Utilities and Data Structures**
    Surveys XVT-Power++'s utility classes, which serve as a link between different XVT-Power++ features and the XVT Portability Toolkit, and the limited but useful data structure classes that XVT-Power++ uses heavily and makes available to the application developer.

Appendix A: Error Messages
An appendix that lists the XVT-Power++ error messages according to the classes that generate them. The classes are presented in alphabetical order, and the error messages pertaining to each class are listed numerically according to the numbers XVT-Power++ assigns them

How to Read This Manual

If you are a first-time XVT-Power++ user, we recommend that you read the first two chapters of this manual to get an overview of XVT-Power++ and an understanding of the XVT-Power++ coding conventions and style guidelines. Then work your way through the tutorial in Chapter 3. Chapters 4 through 7 explain how XVT-Power++'s application framework is structured and present information that every XVT-Power++ user should know. Once you have finished the first seven chapters of this manual, you can read the others as your needs and interests dictate. Keep in mind that you can always consult the *XVT-Power++ Reference* for a detailed discussion of any class in the XVT-Power++ hierarchy.

1

XVT-POWER++ OVERVIEW

1.1. Introduction

This chapter gives you an overview of the roles assigned to different parts of the XVT-Power++ system so that you will know how the various components interact with one another. It considers these roles in terms of the specific development tasks that you may perform. It also discusses the advantages of using XVT-Power++'s object-oriented approach to design an application.

1.2. What's in the XVT-Power++ Package?

The classes in XVT-Power++ fall into three major categories: application framework, utilities, and data structures. This section surveys the functionality that XVT-Power++ makes available to a GUI application developer within these categories—in terms of specific development tasks.

1.3. Application Framework

The XVT-Power++ application framework consists of three different levels:

- An *application level* that controls a program and is analogous to `main`
- A *document level* that gets access to data and stores and manages data
- A *view level* that provides windows and other specialized structures in which to display data and graphical objects

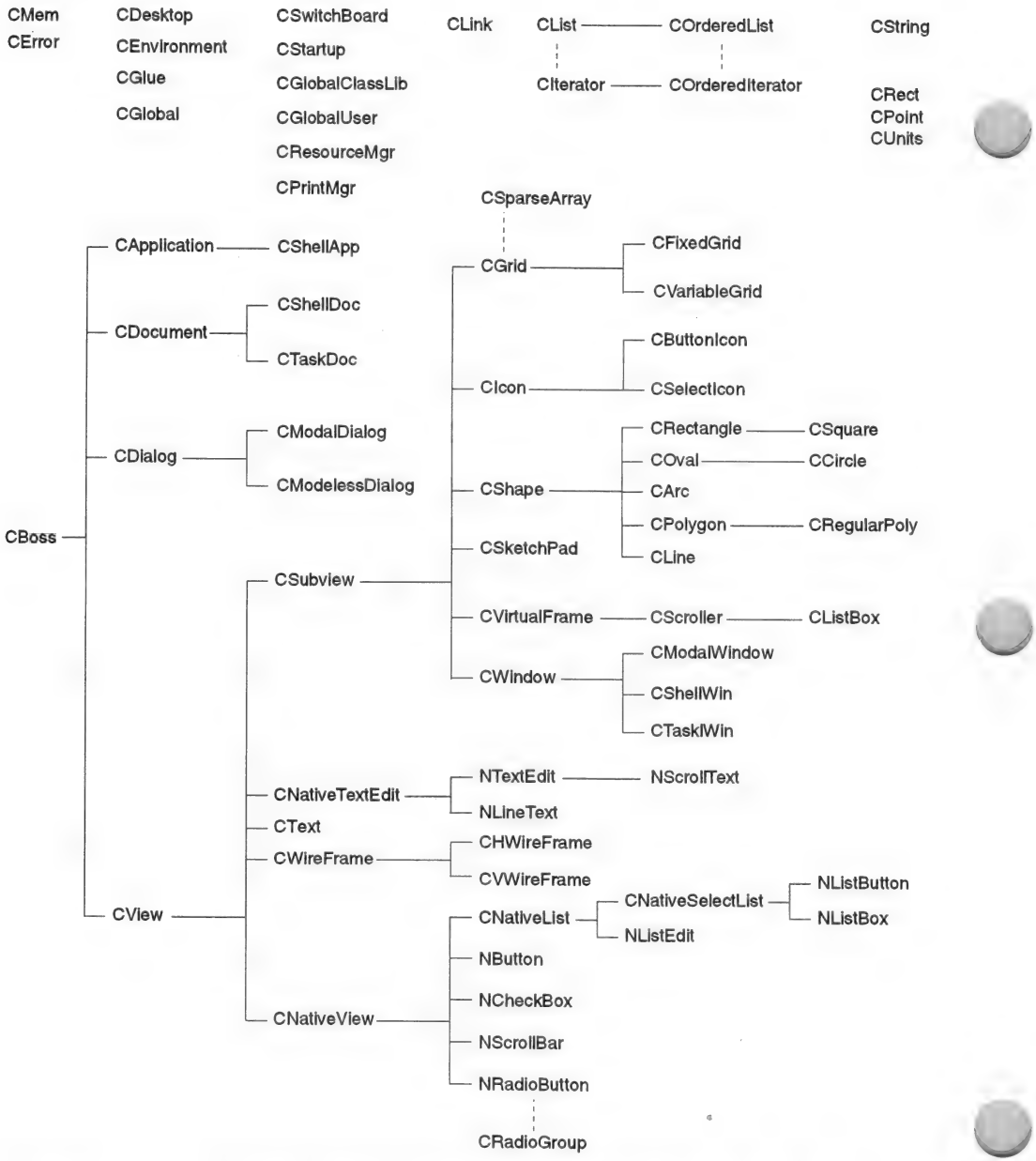


Figure 1. The XVT-Power++ Class Hierarchy

1.3.1. Application Level

The application level controls various aspects of the program: starting it and shutting it down and initializing the network connections, database connections, and any other connections needed by the application. Once the `CApplication` object is instantiated, it sets up any default menus and may put up a menubar, display a splash screen, or display an About window. It has basic functions for managing and creating documents and for communicating with different objects within the application through the use of XVT-Power++'s commands and messages. It also provides some basic means of communicating with the user, such as bringing up dialog boxes to open files, set up printing, create a document, add a document, notify all documents to close, and indicate which document has a given piece of data.

The following development tasks are handled at the application level:

1.3.1.1. Controlling the Program

Each XVT-Power++ application has one `CApplication` object that takes the thread of control when the program is started. This object is the first one to be instantiated and the last one to be destroyed. `CApplication` is an abstract class from which you derive the specific application object for your program.

1.3.1.2. Handling Application Startup

Your program transfers control to XVT-Power++ inside of `main`. To start your program, giving control to XVT-Power++, create an instance of your user-derived CApplication class and invoke the `Go` method. An example of this procedure, found in the **CStartUp.cxx** file, is shown here.

```
void main(int argc, char* argv[])
{
    CMyApp theApplication;
    theApplication.Go(argc, argv, MY_MENU_BAR_RID,
        ABOUTBOX, "BaseName", "Application Name",
        "Task Title");
}
```

No code should follow the call to `Go` since XVT-Power++ never returns control to your program once the `Go` message is processed.

1.3.1.3. Handling Application Cleanup

The `CApplication` class has a virtual `Shutdown` method that is invoked during the termination of the program. You can override this method to handle any cleanup required by your application.

1.3.1.4. Providing Global Objects and Global Data

XVT-Power++ provides `CGlobalUser`, an optional place for you to insert your own references to global objects, flags, or attributes. If you use this class, keep in mind that it is instantiated by your user-defined `CApplication` object and is processed through the `CApplication` initializer.

Another class, `CGlobalClassLib`, contains the global variables for the class library. This class is privately defined, and you should not add any application-specific information to it.

One example of a global object is the XVT-Power++ desktop. It is a class in charge of managing the windows on the screen—their placement, stacking order, and so on. Each application has one desktop object that is global to everything and that is activated through `CGlobalClassLib`.

1.3.1.5. Getting Access to Global Objects and Global Data

Your application gets access to all global data through CBoss, which is initialized through the CApplication class. The CApplication initializer allows you to set up the global user for your user-derived application object.

`CBoss` contains two static pointers, one for user-supplied globals (`*GU`) and one for XVT-Power++ globals (`*G`). Anything that inherits through `CBoss` has access to the global data through one of these pointers. You must set each of these pointers to an actual object. When a `CGlobalUser` object is created, the `CBoss` initializer informs `CBoss` of the existence of the global utilities for the user application.

1.3.1.6. Finding Out About Global Definitions in XVT-Power++

The class that contains global definitions for XVT-Power++ is CGlobal, which is actually a file. You may need to refer to it occasionally to find out how something is defined; do not modify this file. Consult CGlobal when you need to know about glue types, default parameters, internal XVT-Power++ commands, and so on. You may need to know what resources XVT-Power++ defines internally or what the XVT-Power++ ID number base is. When you

want to take a look at the file, see the section on CGlobal in the *XVT-Power++ Reference*.

1.3.1.7. Creating Documents

Your user-defined CApplication object is responsible for instantiating one or more user-defined and derived CDocument objects. For examples of how to overload the methods on CApplication and CDocument, see CShellApp and CShellDoc in the *XVT-Power++Reference*.

1.3.1.8. Propagating Messages from One Class to Another

CBoss, which is never itself instantiated, supplies the basis for the event and message-passing structure. It has three methods for event hooks that are located inside objects throughout the application framework hierarchy: DoCommand, DoMenuCommand, and ChangeFont.

DoCommands can be passed up the entire hierarchy, from the deepest subview on up to the window, from the window to the document, and finally from the document up to the application. It is very important when you overload a DoCommand that it call the inherited DoCommand by default, which in turn calls the CApplication object's DoCommand. Calling the inherited DoCommand as a default permits the propagation of data. This applies in all DoCommand cases.

1.3.1.9. Creating a Desktop to Manage Screen Window Layout

The user-derived CApplication object creates one CDesktop object per application. All core XVT-Power++ classes have access to the desktop through the global references stored by CBoss::G (G -> GetDesktop).

1.3.1.10. Setting Up Menus and Handling Menu Commands

Both CApplication and CDocument contain SetUpMenus method hooks that you must override to provide information on setting up the initial menu defaults. From CBoss, these classes inherit a DoMenuCommand method that they override.

1.3.1.11. Getting Examples of How to Override a Class

The shell classes, CShellApp, CShellDoc, and CShellWin, are overrides of CApplication, CDocument, and CWindow, respectively. You can consult them as examples in the *XVT-Power++ Reference*.

1.3.2. Document Level

The document level of XVT-Power++'s application framework is responsible for accessing and managing data. The `CDocument` object manipulates files or internal pieces of data and acts as the link between the application and the views of the data. A document cannot itself display data, so it instantiates a window in which to view the data.

The following development tasks are handled at the document level:

1.3.2.1. Getting Access to Data

Objects of the `CDocument` class are responsible for providing access to the data to be displayed inside views, in the form of files, records, hooks to a database, and so on.

How a document obtains its data is up to the individual application. If a document is to be in charge of a file, it must be able to open a file, read and write from it, and close it. If a document needs information from a database, it must be able to make the connections to the database, select from it, commit changes to the database, and so on. You are responsible for writing the data access methods. You may choose to put some of the code for obtaining and setting up data in the `CDocument` constructor—if there is not much initial setup to do or if the document is going to use the data upon creation.

Even after the document is created, you can delay the obtaining of data until a method of the document is called. This depends on how you design your documents. Basically, keep in mind that you must add methods that obtain data, and these methods should be called at different points by the application. A `CDocument`-derived class could override the `CDocument DoOpen` method so that it is called by the application after it creates the document or when a user selects Open from the menubar. For an example, take a look at the `DoOpen` and `DoNew` methods on `CShellDoc`. For specific information on how a document gets access to data, see Section 3.3.

1.3.2.2. Creating Windows

Objects of the `CDocument` class are responsible for instantiating and managing a common set of windows. Group your application windows by functionality or other similarities and use one `CDocument` object per group to create and manage the windows. The `BuildWindow` method of `CDocument` is where you put the code that instantiates a window. The `BuildWindow` mechanism can be called from `DoOpen` or `DoNew`.

1.3.2.3. Creating Variations on the Basic Window

Inheriting from XVT-Power++'s basic window class, `CWindow`, are three variant child classes: `CModalWindow`, `CShellWin`, and `CTaskWin`. All three classes not only provide additional window functionality but also serve as examples of how classes are derived from CWindow. The first, CModalWindow, gives modal behavior to a window. When the window is opened, it takes over the screen and disables all other windows so that nothing else can happen while it is open. Modal windows are useful when your application needs an item of information or a commitment from the user before it can continue. `CModalWindows` inherit all the properties of `CWindow`, including the ability to nest several different kinds of XVT-Power++ views.

`CShellWin` provides a shell window and is part of XVT-Power++'s shell utility, which is discussed in Section 7.9. Finally, `CTaskWin` is created for use on platforms that require a task window to enclose all other windows in the application, such as Windows and Presentation Manager.

1.3.2.4. Creating Dialog Windows

XVT-Power++ has three classes for creating dialog boxes: `CDialog` (an abstract class), `CModalDialog`, and `CModelessDialog`. In XVT-Power++, the main difference between dialog windows and regular windows are that all dialogs are defined in URL resource format and thus do not inherit the properties of `CWindow`, such as the ability to nest other objects. Dialogs can contain only objects that are defined as controls in a URL resource file. `CModalDialog` provides a dialog window that takes over the screen when it is invoked, disabling all other objects and operations until the user clicks on the necessary button or responds to it in some other way that is indicated. In contrast, `CModelessDialog` does not take over the screen and can go into the background when another window or dialog is brought to the front.

1.3.2.5. Managing Data

A `CDocument` object serves as a central means of communication for changes and updates in different windows. It has hooks for saving data, printing it, closing it, and so on. Currently, you are responsible for writing the code that updates data. Updating takes place through `DoCommand` calls that you can write so that they update all of a document or only part of it:

```
virtual void DoCommand(long theCommand, void*
theData=NULL);
```

1.3.3. View Level

The view hierarchy, the most extensive branch of XVT-Power++, comprises all classes that display some form of object on the screen when they are instantiated. The parent of all these classes is `CView`, an abstract class.

1.3.3.1. Displaying Data

Any class that inherits from `CView` can display itself, including `CWindow`, which is the link between the views and the documents. `CWindow` is responsible for displaying data.

`CWindow` is an abstract class; `CShellWin` is an example of a class that has been derived from it. The window is the topmost view in the nesting of views. Each `CWindow` object corresponds to an actual window on the screen. A window object can be of any XVT type and receives all window events. Most of the window management, such as moving and sizing, is done by the window manager or the XVT-Power++ desktop.

See Also: For information on the XVT window types, see the section on `CWindows` in the *XVT-Power++ Reference* or the Windows chapter in the *XVT Programmer's Guide*.

1.3.3.2. Supplying Native Controls

`CNativeView` is an abstract class from which several different types of controls have been derived, among them buttons, check boxes, scrollbars, list edits, list buttons, list boxes, and radio buttons. When these classes are instantiated, they take the look-and-feel of the native window system in which the application is running.

Native views are the means of communication between the application and the user who is operating the mouse. The user performs such operations as:

- Clicking on a button to start an event
- Scrolling by clicking on a scrollbar
- Making selections by clicking on check boxes
- Making different choices by clicking on radio buttons
- Selecting an item from a list box

While native views can be placed inside views, they cannot contain any views.

1.3.3.3. Nesting One View Within Another

`CSubview` and its subclasses can nest views recursively within other views. This is the basic property that distinguishes them from other classes in the `CView` hierarchy. `CSubview` classes can propagate events to all of their nested subviews. Subviews can have a *selected view* or *selected key focus*, which is the first (and possibly only) subview to receive a certain kind of event.

1.3.3.4. Creating Icons

You can display your own bitmap drawings as icons inside other views. You can even nest other views inside an icon, which is a subclass of `CSubview`. In addition to the basic icon class, there are two derived classes, one for creating an icon that behaves as a select box and one that behaves as a check box.

1.3.3.5. Drawing Shapes Such as Arcs, Ovals, Circles, Squares, Rectangles, Lines, and Polygons

The shape hierarchy, probably the most self-explanatory of the XVT-Power++ view hierarchies, includes squares, circles, polygons, rectangles, and other basic shapes that a user can draw. Some of the objects can be rotated. Application designers can use the shapes to decorate windows and other objects or to communicate with the user pictorially.

Like all *objects* in the frame hierarchy, the XVT-Power++ shapes can receive and communicate events. Like all *views*, shapes can generate automatic `DoCommands` when the user clicks the mouse on them. Thus, they are useful not only for purposes of decoration but also as building blocks to create other objects, such as new buttons. Several of the shapes are very easy to create. For example, you can create:

- a polygon simply by giving a number of sides.
- an arc by giving starting and ending positions.
- a circle by giving a center and a radius.

Suppose that you want a stop sign shape that can act as a button. You would create an octagon using the regular polygon class, nest within this shape a piece of text that says "STOP," and then give it a command number indicating that you want it to act as a Stop button.

Now you have a new button object that allows the user to Interact With Your Application.

1.3.3.6. Creating Grids with Cells of Either Fixed or Variable Size

A grid object is a grouping of cells that are arranged into rows and columns. XVT-Power++ offers three grid classes: the abstract class `CGrid`, and the classes `CFixedGrid` and `CVariableGrid` that allow you to create grids with either fixed or variable-sized cells, respectively. For a color palette window in which you want all squares representing color selections to be the same size, you would use the fixed grid. In a spreadsheet, however, you would use the variable grid so that the columns can have different widths.

A grid has these characteristics:

- It can be either visible or invisible
- It is placed inside a view or window
- It enforces sizing and clipping of items

Grids are useful in CAD types of applications or generally in designs where you want to set the granularity of placement within the drawing area or designing board to a set of grid cells rather than to pixel locations. An item inserted into a grid snaps to a certain corner of its cell and can be centered within the cell; objects snap as they are moved or sized within a cell.

1.3.3.7. Displaying Lists of Selectable Items in a Scrollable Box/ Window

Use the `CListBox` class to display a scrollable box containing a list of selectable text items. The list box is used for choosing from a listing of directories and for navigating among directories.

1.3.3.8. Providing Text Editing Facilities

`CNativeTextEdit`, an abstract class, and its subclasses provide the bulk of the XVT-Power++ text editing functionality: a one-line text area, a variable-sized text editing area, and a scrolling text area.

`CNativeTextEdit` objects can be nested inside a subview, clipped, or hidden. This class has easy-to-use methods for getting and setting part or all of the regular and selected text. It supports quick selection (selection of all text upon any event, such as clicking once, backspacing, and so on). `CNativeTextEdit` provides text validation hooks upon key events.

Below CNativeTextEdit, the text editing tree branches out into variations of text editing objects:

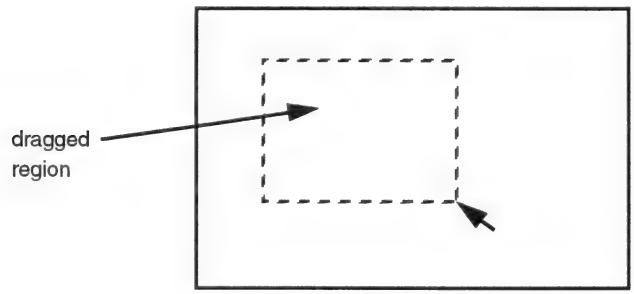
- NLineText — Commonly used for one-line text entries, such as a place for users to enter their names or passwords.
- NTextEdit — A variable-size text editing area in which you can set several attributes, to scroll or not to scroll, for example.
- NScrollText — Provides scrollbars and automatic scrolling.

In addition, there is CText, a static text drawing class that supports control characters such as tabs, carriage returns, and line feeds. It is useful for one-line instructions, titles, and button names.

You can always use the more flexible NTextEdit class in read-only mode to display single lines of text inside a window or subview. However, while NTextEdit is more flexible than CText, it also carries more baggage with it than you may want for one-line displays of read-only text.

1.3.3.9. Designating an Area of the Screen as a Sketching Area

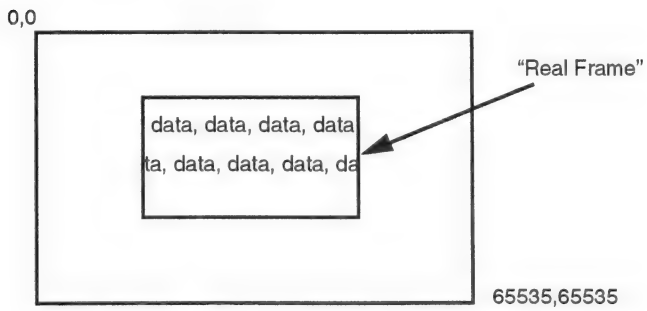
The CSketchPad class is provided for interaction with the user who wants to dynamically draw or create new objects inside a window. CSketchPad can be a basis for CASE or drawing programs. Like most drawing programs, it has an area—called a sketchpad in XVT-Power++—in which the user can drag the mouse to sketch out or draw such shapes as rectangles, circles, lines, and so on. The user can drag out a region to select multiple objects within that region.



1.3.3.10. Representing an Area on the Screen with a Virtual Size Larger Than its Display Area

`CVirtualFrame` is an abstract class that has two regions associated with it: a real, visible region that is located inside a window or some other view and a virtual region. Its subclass, `CScroller`, represents a virtual frame with scrollbars attached to the viewing area.

Virtual frames are created for cases when the screen is not large enough to display all of the viewable information at once. The virtual frame allows you to create subviews that are of any virtual size you want—3,000 pixels by 5,000 pixels, for example. The information is actually displayed on the screen inside the real frame, which can be much smaller than the virtual frame. Only a certain area of the viewable information can be viewed through the real frame at one time.



The user views the different areas of the virtual frame by scrolling the real frame up and down. `CVirtualFrame` is used in conjunction with the `CScroller` class, which attaches scrollbars, either vertical or horizontal or both, to the real frame area. Thus, the user can scroll the virtual frame around to view whatever displayable text resides inside the virtual area.

1.3.3.11. Creating a Rubberband Frame for Sizing and Dragging Objects

To get a rubberband frame that surrounds a `CView` object, enabling it to be dragged or sized with the mouse, use `CWireFrame`. This `CView` class acts as a helper to all other `CView` classes. Typically, you do not need to be aware of `CWireFrame` since it is used internally. However, you can override this class and modify the way the wire frame draws or alter the way the rubberbanding is implemented.

1.3.3.12. Attaching scrollbars to a Window or View

XVT-Power++ contains four classes that pertain to scrollbars. `NScrollBar` provides a horizontal and/or vertical scrollbar that has the look-and-feel of scrollbars in the native window manager. `CScroller` attaches scrollbars to the visible region of a virtual frame. `NScrollText` attaches scrollbars and autoscrolling support to text editing boxes. Finally, `NWinScrollBar` provides scrollbars that are instantiated internally when windows with scrollbars are created. These scrollbars are part of the window itself, whereas `NScrollBars` are not attached to the window and can be resized or moved around inside the window.

1.3.3.13. Resizing a View While Retaining Its Proportions/ Placing a View and Specifying How It Can Be Resized

The `CGlue` class gives stickiness properties to objects. If an object has an associated `CGlue` object, then its stickiness properties enable it to stay fixed by a constant distance from the borders of its enclosure. There are glue types for sticking an object to the bottom, top, left, right, bottom right, top left, top right or bottom left of a view—or over all of the view.

1.3.4. Utility Classes

XVT-Power++'s *utility* classes provide links between different parts of the XVT-Power++ system, such as events and the application, the XVT-Power++ library and your specific application, global objects and the rest of the objects in the hierarchy, the screen window layout in your application (the desktop), drawing tools and your application (the environment), and resources and your application.

Each of XVT-Power++'s main objects has several helping and utility objects attached to it. For example, a native view can have a glue object, an environment object, a title object, and a list of commands. Every view has a `CRect` object that tells it where to draw and a `CPoint` object that tells it which origin to draw from and what its coordinates are relative to.

1.3.4.1. Storing Program Resources Such as Icons and Text Strings

Almost all resources used by XVT-Power++ applications can be coded using XVT Portability Toolkit's URL and the CURL compiler. However, for applications running on X platforms, the resources must be coded separately in `CResourceMgr`. Use

CResourceMgr to indicate what resources your application will use. A resource can be a bitmap, an icon, the resource format of dialog boxes or strings, and so on. Anything that your application uses can be stored, managed, and created by the resource manager. For example, a stop sign icon for a STOP button would be recorded in the resource manager file as a resource for the application. The resource manager then knows that the icon resource exists, and when there is a call for it, it can draw the icon.

1.3.4.2. Defining Window and View Foreground and Background Colors, Font Types, Drawing Modes, and Line Colors and Widths

The CEnvironment class provides a data structure containing information on such environment attributes as colors, foreground and background, pen, brush, and drawing mode. Environment information can be propagated down the object hierarchy. By default there is a global environment object that is shared by every displayable object in an XVT-Power++ application.

1.3.4.3. Reporting Errors

XVT-Power++'s error reporting facility is provided in the CError class.

1.3.4.4. Allocating Memory

XVT-Power++'s CMem class handles memory allocation.

1.3.4.5. Translating The XVT Portability Toolkit Events to XVT-Power++ Calls

CSwitchBoard serves as a liaison between the XVT Portability Toolkit and XVT-Power++. The switchboard is in charge of channeling whatever events are happening in the system to the appropriate object. For example:

- A termination or startup event goes to the application object
- A resize event goes to the appropriate window object
- A key event is channeled by the switchboard to the appropriate view

1.3.4.6. Debugging a Program

XVT-Power++'s CMem class provides some memory management debugging utilities. (Additional debugging utilities are planned.)

1.3.4.7. Printing

XVT-Power++'s interface to the XVT Portability Toolkit's printing facilities is called `CPrintMgr`. This class is in charge of queuing up data and printing it. Normally, when you want to print a view, you can call `DoPrint` inside the view. The actual implementation of printing is handled inside `CPrintMgr`.

1.3.5. Data Structures

XVT-Power++ contains hierarchies of data structures and will continue to add more in future releases. Most of the classes already discussed make use of these structures, which are generally *container* classes that are used to store objects. Users often need to store objects within data structures. For example, XVT-Power++ itself makes heavy use of lists.

1.3.5.1. Specifying Locations on the Screen

Every XVT-Power++ object has a `CPoint` object that tells it which origin to draw from and what its coordinates are relative to. Each `CPoint` is an *x,y* coordinate.

1.3.5.2. Storing Sets of Coordinates/Placing Views on the Screen

Every XVT-Power++ object has a `CRect` object that tells it where to draw. A `CRect` is a rectangular region (set of coordinates) on the screen that defines an area in which another object will be located. `CRect` has methods for translation, coordinate system conversion, intersection, union, inflation, height, and width.

1.3.5.3. Converting Global to Local Coordinates and Vice Versa

The `CRect` class has several methods for coordinate system conversion. These methods allow you to go back and forth between global and local coordinates without having to do any calculation.

1.3.5.4. Specifying Logical Units for Defining Screen Coordinates

By default, all XVT-Power++ applications use a one-to-one pixel mapping for drawing or printing. You can set a different mapping—in centimeters, inches, characters, or a user-defined unit—by instantiating a `CUnits` object and then calling a certain object in the application framework hierarchy and setting its units through its `SetUnits` method. The use of logical units makes applications more portable because different machines have different screen widths, heights, and resolutions. The logical units are mapped out to the

physical device on which you are displaying your information: either the screen or a printer.

1.3.5.5. Representing Character Strings

`CString` is a class representation of character strings that encapsulates the data structures and utilities of character strings.

1.3.5.6. Concatenating and Appending Character Strings

The `CString` class contains several methods for concatenating and appending character strings.

1.3.5.7. Comparing Character Strings

The `CString` class contains several methods for comparing character strings. It enables you to do equality, inequality, greater-than, less-than, greater-than-or-equal and less-than-or-equal comparisons.

1.3.5.8. Storing Items in Lists

XVT-Power++'s `CList` class stores references to generic objects according to their addresses. You cannot assume any order for storing items or any means of storage. Items can be multiply referenced.

1.3.5.9. Iterating Over Lists

`CIterator` is a class that iterates over `CList` objects. You can assume no specific iterating order.

1.3.5.10. Linking Items in a List

`CLink` is a `CList` helper class that represents each of the links in the chain of a list. Each link points to the previous and next links in the list and to the item in the link.

1.3.5.11. Ordering a List

`COrderedList` is similar to `CList`, except that the items in the list have an order; a position is associated with each item on the list. A helper class, `COrderedIterator`, iterates through the elements of the list in ascending position order.

1.3.5.12. Storing Two-dimensional Arrays and Conserving Memory

`CSparseArray` is useful when the data stored is sparse because the number of cells in its two-dimensional array is equal to the number of nonempty array locations.

1.4. Designing an XVT-Power++ Application

When you begin to design an application, you must make several basic decisions, such as what it will look like, what will happen when the user executes the program, what will come up first, and the order of the steps that the user must perform to work the application. XVT-Power++'s application framework helps you to make these decisions. When you design an application, you will follow these basic steps:

1. At the very least, write a new application, a new document, and a new window class, deriving them from their corresponding XVT-Power++ classes.
2. Determine what XVT-Power++ classes you want to use.
3. Derive/Write other classes that reflect the things you want your application to do.
4. Design the structure of your own class and object hierarchies and decide how to assign the *supervisor* relationships among these classes. Who is going to own whom? Who is going to instantiate what?

Your application will have the same skeleton, the same underlying object hierarchy, as XVT-Power++, but it will also have its own specific flavor.

1.4.1. Development Platform

Naturally, the platform on which you develop an application has an impact on your design decisions—imposing constraints, for example, on what is going to come up first and how it will look. On the Macintosh, when the user executes a program, a menubar appears, and the user is expected to do a *new* or *open* operation. This is also true in Motif. In MS-Windows, an application always starts up with one window on the screen, the task window. Whatever the platform, when an application starts up, something appears on the screen for the user to see. It acts as a clear visual cue about what to do next in order to gain entry to the program's functionality.

XVT-Power++ automatically adapts to the native look-and-feel of the platform for which it is compiled. Thus, it creates the appropriate windows and menubars upon application startup.

1.4.2. Advantages of Object Hierarchies

Class hierarchies are very useful structures because a class can inherit a lot of features from a parent class, allowing developers to reuse some of the code inside a parent class. We can reuse this code because classes inherit down the line.

This is similarly true of object hierarchies, though for a different reason. We illustrate this point with the list box, which is a composite of several objects. If you look at a list box on the screen, you'll notice that it has horizontal and vertical scrollbars, which are instantiations of XVT-Power++'s `CScroller` class. More hidden to you is the fact that the list box contains a grid so that all the objects in the list box can easily be aligned. Inserted inside the grid are the actual text or string items, `CText` objects.

Although `CListBox`, as a composite, is a fairly complex class, it was easy to implement because the objects composing it already exist in XVT-Power++—scrollers, grids for formatting, and text objects. Thus, all we have to do is put it all together and add a few specifics that list boxes require in order to manage the list: insertion and removal of items, selection and deselection, reporting which items are selected, and so forth.

1.4.2.1. Advantages for XVT-Power++

In the `CListbox` class, XVT-Power++ reaps one of the great advantages of object-oriented programming: the ability to put together new objects out of objects that have already been written. Through the reuse of code and the reuse of ideas,we easily implemented this new composite object.

1.4.2.2. Advantages for XVT-Power++ Users

an XVT-Power++ user does not need to know that a `CListbox` is a composite of the `NScrollBar`, `CList`, and `CGrid` classes. When the user instantiates a list box, its behaviors are already built into it. To get complete list box functionality, a user simply creates a new list box (`new CListbox`) and passes it some parameters that give it a location, strings to insert into it, and so on. Then the list box appears inside the view, fully functioning. However, if you are an application designer who wants to take full advantage of

XVT-Power++ and derive your own classes, then you need to know how such composite objects as the list box are put together so you can create your own.

1.4.2.3. Advantages for Designers of XVT-Power++ Applications

XVT-Power++ contains a wide range of classes that provide most of the functionality needed in a typical GUI application. However, when designing an XVT-Power++ application, you may find it necessary to write a completely new class from scratch. Most of the time, when XVT-Power++ does not supply a specific class or behavior that you need, you can build a composite object from classes that XVT-Power++ already provides. This is the essence of object hierarchies, and it is what makes XVT-Power++ extensible and very easy to use.

There may also be times when you want to alter the behavior of an XVT-Power++ class. To create a new class, derive it from the class you want to change and then override the appropriate method(s). For example, if you wanted a list box that contains graphical objects instead of text objects, you would derive it from XVT-Power++'s `CListbox` class and modify one of the methods after studying the class to find out what can be overridden and what cannot. You discover that you can override the method `InsertItem`. In another case you might have had to *add* a method that takes graphical objects and inserts them into the grid.

In overriding the method on `CListbox`, you are creating a completely new class—say, `MyListbox`—leaving the original `CListbox` class intact. You should never change the actual code of an XVT-Power++ class. The class library is designed so that you do not change the code if you want to modify behavior. Instead, you derive a new class and override one or more of the methods on the XVT-Power++ class. The XVT-Power++ shell classes—`CShellApp`, `CShellDoc`, and `CShellWin`—are examples of derived classes. See the *XVT-Power++ Reference* for detailed information on these classes.

1.5. Where To Go Next

At this point, we have covered the functionality that XVT-Power++ makes available in:

- The `CBoss` hierarchy, which is the main set of classes in the XVT-Power++ application framework. This hierarchy provides the view classes and is the hierarchy to which we

will be adding classes to do data management. All communication layers are contained within this hierarchy

- The *utility* classes or files that provide links between different parts of the XVT-Power++ system
- The *data structure* hierarchy, which provides the means to create the data structures such as lists and arrays

Now that you have surveyed the XVT-Power++ classes and are aware of the overall functionality available to the XVT-Power++ application developer, we recommend that you read about style guidelines and coding conventions in Chapter 2, and then work through the tutorial in Chapter 3. For detailed information on any of the classes mentioned in this chapter, consult the *XVT-Power++ Reference*.

2

CODING CONVENTIONS AND STYLE GUIDELINES

This chapter presents the coding conventions and language/style guidelines that XVT-Power++ follows. Awareness of these guidelines will help you use XVT-Power++ more efficiently.

2.1. File Structure

With few exceptions, XVT-Power++ consists of a set of C++ classes. The most basic rule is that there must be one class per file. The name of a file matches the name of the class to which it pertains. Each class has a pair of files, a **.h** (header) and a **.cxx** (source) file. Thus, a class named `CApplication` would be stored in two files: **CApplication.cxx** and **CApplication.h**. The **.h** file contains the definition of the class and any other information that the class needs in order to be accessed by users of the class. The **.cxx** file is the source file; it contains the actual functions and methods for the class.

Occasionally, a header file will contain more than one class. In such a case, the classes must be very closely related. One is a helper class for the other.

2.1.1. Including Files for Usage

When you write a piece of code that uses a certain class, you must include that class's definition. The name of the file containing the definition of the class may vary from platform to platform. For example, on some platforms `CRegularPoly` is stored in a file named **CRegularPoly.h**. However, for applications running on Windows, this is an invalid filename because DOS restricts filenames to only eight characters and a three-character extension.

XVT-Power++ provides a special structure for including files that allows you to name your files as you desire, without worrying about platform restrictions. This structure is illustrated as follows:

```
#include "PwrDef.h"

#include CRegularPoly_i
#include CCircle_i
#include CScroller_i
#include CRect_i
```

In the case shown here a piece of code uses the `CRegularPoly`, `CCircle`, `CScroller`, and `CRect` classes and thus needs the definitions of those classes. Each class name with the `_i` appended to it is a macro that finds the appropriate file containing the class definition. Before you can include the classes, you must include the **Pwref.h** file, which defines the macros.

2.2. Naming Conventions

This section describes XVT-Power++ naming conventions. It is not exhaustive, but it does cover the most common cases. Mangling is also discussed.

2.2.1. Classes

Class names begin with a prefix letter, a `C` for most classes. XVT-Power++ requires capital letters rather than underscores as separators. That is, the prefix and the first letter of each word in the class name are capitalized. For example:

- `CApplication`
- `CRadioGroup`

The native classes use the prefix `N`. For example:

- `NButton`
- `NScrollBar`

2.2.2. Data Members

Class data members use a lowercase prefix of `its` or `it`, as follows:

- `itsData`
- `itIsSelected`

2.2.3. Methods

The initial letter of each word in a method name is capitalized, as follows:

- `Draw`
- `DoDraw`

`Event` methods are handled by the view object itself, while `DoEvent` methods are both handled and passed down to the rest of the subviews, which, in turn, pass them on down to any subviews they may contain. For example:

- `Draw` — draw this view object.
- `DoDraw` — draw this view object *and* inform all of its subviews to draw.

2.2.4. Class Statics

Static class methods or data members take as a prefix the name of the class to which they belong. Also, constants appear in capitalized letters (all-caps), as shown here:

```
CWireFrame::WIRESIZE // constant static
CWindow::itsNumberOfControls // not constant
CWindow::GetNumberOfControls() // a static method
```

2.2.5. Constants and Defines

The first word of a constant appears in all-caps. Any other words must appear in either all-caps or initial caps. For example:

```
NULL
NULLIcon
MAXSize
```

2.2.6. Functions

Functions are not treated differently from methods. That is, the first letter of each word in the function name is capitalized:

```
Foo
PrintMessage
```

Within the signature of a method or a function, the parameters have a prefix of `the` to distinguish them from local variables:

```
theNewRegion
theData
```

2.2.7. Variables

Local variables have a prefix of `a` or `an` to distinguish them from parameters for methods and functions:

```
aRect
anEnvironment
```

XVT Portability Toolkit-related names contain the word XVT, as follows:

```
GetXVTWindow
itsXVTControl
```

Type	Rule	Example
CLASS NAME:	[PREFIX+[Word]]	CRect
DATA MEMBERS:	[its+[Word]]	itsPoint
STATIC MEMBER	[class::its[Word]]	CWindow::itsPlatform
METHODS:	[[Word]]	DoSomething
VARIABLE:	[word+[Word]]	charCounter
OBJECT:	[a+[Word]]	aPoint
XVT OBJECT:	[xvt+[Word]]	xvtWindow
PARAMETER:	[the+[Word]]	theItem
#define:	[WORD]	COMPILER
const:	[k+[Word]]	kPi
C++ FILE:	[*PREFIX*+[*Word*]].[h|cxx]	CRect.cxx

2.3. Mangling

Mangling is a useful utility which ensures that the names of XVT-Power++ classes will not clash with the names of any other classes that you define. For example, let's consider the XVT-Power++ class `CString`. Perhaps you are also using some other class library that, by coincidence, also contains a class named `CString`. The compiler and linker must be able to tell whether the term "`CString`" refers to the XVT-Power++ class of that name or to another party's `CString` class.

By default, XVT-Power++ mangles the names of all of its classes by giving them a prefix of `Pwr_`. At compile time, XVT-Power++ converts its class named `CString` to `Pwr_CString`. The XVT-Power++ library does not have a definition for `CString`;

rather, it has a definition for `Pwr_CString`. Thus, you do not usually have to worry about name clashes.

Suppose you are in a file, `foo.cxx`, in which you want to use both the XVT-Power++ `CString` class and another party's `CString` class. Immediately following the inclusion of the string header file, you would put in the following line:

```
#undefine CString
```

In the rest of the file, you must explicitly put the mangling prefix on the XVT-Power++ `CString` class. `CString` will now refer to the other `CString` class, and `Pwr_CString` will refer to XVT-Power++'s string class.

Example:

```
// Include Pwrfiles:
#include ...
#include CString_i
#include ...

#undefine CString

// Include Non PwrFiles
#include ...
#include "SomeOtherCString.h"
#include ...

// Code:

void foo(void)
{
        CString s1 = "This is a non-Pwrstring";
        Pwr_CString s2 = " This is a Pwr-string";
}

// Now redefine CString to be a Pwrstring:

#define CString Pwr_CString

void goo(void)

{
        CString s1 = "This is a Pwrstring";
        CString s2 = "This too is a Pwrstring";
}
```

2.4. C++ Style Guidelines

This section presents C++ style guidelines that XVT-Power++ users follow. Understanding them will enable you to use XVT-Power++

more efficiently; following them will make your code more compatible with XVT-Power++.

2.4.1. Const and Enum

Use `const` and `enum` rather than `#define` whenever possible, allowing your programs to take full advantage of C++'s type safety.

```
const float pi = 3.14159;
typedef enum {TRUE = 1, FALSE = 0} BOOLEAN;
```

Make full use of constant methods whenever applicable, as shown here:

```
CRect CWindow::GetFrame(void) const;
```

A caller to `GetFrame` is assured that this call will not change the state of the object through which it was called.

2.4.2. Inlines

Separate inlines from the actual class definition to avoid cluttering the interface. Most inline code is placed in a separate include file so that implementation is not revealed when the interface is the only concern. For example:

File 1:

```
// File CRect.h

class CRect
{
    public:
        ...
        CRect Inflate(int theInflation);
        ...
};

#include "CRectInline.h"
```

File 2:

```
// File CRectInline.h

inline CRect CRect::Inflate(int theInflation)
{
    // ... code ...
}
```

2.4.3. Overloaded Methods

XVT-Power++ strives to overload methods only while preserving semantics. For example the following two methods take different parameters but have the same outcome:

```
CCircle::Size(int theNewRadius, const CPoint&
theNewCenter);

CCircle::Size(const CRect& theNewBoundingRegion);
```

2.4.4. Internal Structure of Classes

Classes are always organized as follows:

```
Class CApple
{
        public:

            // Data:
            // Methods:

        protected:
            // Data:
            // Methods:

        private:
            // Data:
            // Methods:
};
```

When you organize your classes as shown here, users of the class who are interested simply in the interface to the class need only look at the top of the file. They do not have to wander through the entire class definition looking at protected and private information that they cannot call.

2.4.5. Function Parameters

Many times programmers calling methods or functions wonder what they can validly pass into them, whether they need to delete what they pass in later, whether they need to worry about their object being modified or not modified, or whether the object they are passing in will actually be copied into another object. Because all of these questions arise, XVT-Power++ simplifies the number of case situations that can happen for parameters when a function is called.

XVT-Power++ function parameters can have only *one* of the following four combinations. These four flavors represent a progression in the number of things that can happen, from least to most. The non-constant pointer, then, is the case where the most things can happen, and we recommend that you closely read the

documentation on any method taking such parameters in the *XVT-Power++ Programmer's Reference*. Thus, you should look at the type of the function parameter, and, depending on the type, you can make some assumptions about the semantics of the function call.

2.4.5.1. Pass by Value

```
example: void SetId(int theNewId);
```

Normal pass by value semantics can be assumed. Objects of user-defined types are seldom passed in this way. Instead, the syntax of constant references (discussed below) is used. However, note that from the caller's point of view, the semantics of pass by value and XVT-Power++'s use of constant reference are identical.

2.4.5.2. Constant References

```
example: void SetSize(const CRect& theNewSize);
```

Whenever a function takes a constant reference, the user can assume pass by value semantics. The reference is used only for efficiency. Thus, after the call, for example, the new size can be deleted without side effects. In addition, calling `SetSize` twice with two independent yet identical `CRect` objects has the same result each time. The identity of the object pointed to is irrelevant.

2.4.5.3. Constant Pointers

```
example: int GetId(const CView* theView);
```

The address of the object is needed, but the object itself will not be modified. The identity of the object pointed to is important to the method. Furthermore, the function guarantees *not* to store the address of the object for future use. After a call to the method, the object might be destroyed or mutated.

2.4.5.4. Non-constant Pointers

```
example: void SetGlue(CGlue* theNewGlue);
```

The method requires the address of the object. Depending on the method, the object pointed to may be modified or destroyed, or its address might be stored for later usage. It is therefore very important to read the documentation on such a method so that you can use it correctly.

2.4.6. Return Values

The issues here are similar to those for parameters, except that now we are concerned with what is returned. XVT-Power++ returns only the following four values.

2.4.6.1. Temporary Values

```
example: int GetId(void);
CRect GetFrame();
```

On the safest side are the temporary values.These are usually used on the spot or are copied into local variables. Note that returning temporary copies to large objects can be inefficient. XVT-Power++ sometimes avoids this by using reference counting. For example, the following function retrieves a paragraph of text and returns a temporary `CString` object:

```
CString NTextEdit::GetParagraph()
```

Since `CString` is reference counted, making a copy of the entire paragraph into a local object has very little overhead.

2.4.6.2. References

```
example: CRect& operator=(const CRect& theRect);
```

The return value can be used as an `lvalue`. XVT-Power++ guarantees to return references only to the object through which the method is invoked. References to newly allocated objects are never returned. The example shown here makes possible the following assignment:

```
a = b = c;
```

2.4.6.3. Constant Pointers

```
example: const CEnvironment* GetEnvironment(void);
```

The pointer returned points to an object that must not be modified or deleted. The compiler enforces the fact that the object pointed to cannot be changed. Take care not to cast away the constants. The pointer's "validity lifetime" varies, depending on the function.The validity lifetime is the length of time the pointer remains valid. Will the pointer be valid ten function calls from now? That is, will it still be pointing to the same object? To find out how long a pointer will be useful to you, read the documentation on the method in the *XVT-Power++ Reference*.

2.4.6.4. Non-constant Pointers

In this fourth case, simply a pointer is returned. Read the documentation on the method in the *XVT-Power++ Reference* to see what you are and are not allowed to do. Usually, you are allowed to modify what the pointer points to, but be careful about deleting the pointer. XVT-Power++ does not return non-constant pointers if it can avoid doing so.

2.4.7. Inherited Methods

You can assume an `is-a` relationship when one XVT-Power++ class is derived from another. Thus, only public inheritance is used unless otherwise noted in the documentation. Following are two examples of inherited methods:

```
CView::Size(const CRect& theNewRect);
CScrollBar::Size(const CRect& theNewRect);
```

XVT-Power++'s main goal when overriding a method, which should be the goal of all XVT-Power++ users, is to maintain the semantics of a method. The `Size` method of a view has its own semantics, meaning that the view now has a new size, and that is all. If the documentation says that this method simply sizes and resizes the internal structure of the class and does not redraw it, then the scrollbar `Size` method should only size the internal structures and not redraw. Different derivations might change the implementation and take care of some extra things the class has to do in order to deal with the method, but they do not change the semantics or do anything that is not expected of the inherited method.

2.4.8. Basic Class Utility Methods

There are some utility methods present in every XVT-Power++ class. Every class is guaranteed to have a constructor and a destructor. Also, an assignment operator is defined for every class. Many classes allocate some memory and have pointers to other objects, so it is important to override the equal operator to ensure that it is safe. The same is true of the copy constructor, which is overridden everywhere for safety purposes. Many times, usage of the copy constructor is not recommended; sometimes its usage is disabled. Nonetheless, you are guaranteed that your program will not crash because you have used an equal operator or a copy constructor. An additional zero-argument constructor is provided for as many classes as possible. This constructor allows users to do such things as create an array of class objects. However, many

XVT-Power++ classes currently do not have a way to be constructed without any parameters, and thus lack such a constructor.

2.4.9. Templates

XVT-Power++ does not use templates in its current implementation, and it will not use them until more compilers are available that use C++ 3.0, which includes templates. At that time you can expect a movement towards templates, especially in XVT-Power++'s data structure classes.

2.4.10. Reference Counting

XVT-Power++ provides reference counting in its data structure classes, typically `CList`, `COrderedList`, and `CString`. No reference counting is provided for the other XVT-Power++ classes, although it might be in later releases.

3

XVT-POWER++ TUTORIAL

3.1. Introduction

XVT-Power++ is designed to be easy to learn and use. Although the library is made up of over sixty different classes, you can accomplish fairly complex tasks once you know how to use only a few of them. To that end, this tutorial chapter guides you step-by-step through the process of using XVT-Power++ to build three very simple graphical user interface applications. To complete these exercises successfully, you need some understanding of object-oriented programming and a good grasp of C++ and its syntax. Also, you should have read Chapter 1 of this manual.

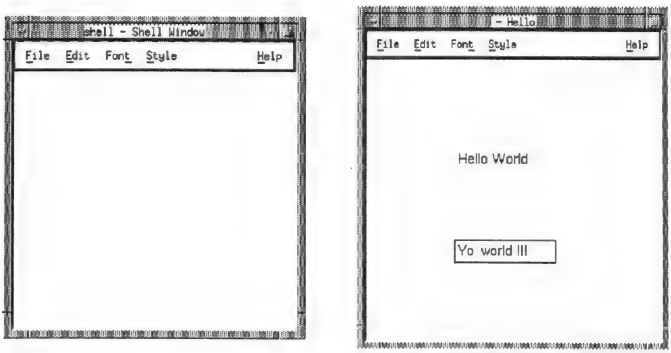
This interactive tutorial has three main sections. We start with a simple "Hello World" exercise that introduces some `CView` classes as well as some of the XVT-Power++ events and messages. In this exercise, you will create an interactive windowing program that can display not only the "Hello World" text string but also other text strings. Next, we explore the use of documents by implementing a simple file editor program. Finally, we guide you through the design and implementation of a drawing program that illustrates the use of XVT-Power++'s entire application framework.

Some of the examples that you will encounter use XVT Portability Toolkit calls. We do not assume that you are familiar with the XVT Portability Toolkit, so we simply point out where XVT Portability Toolkit calls are made to distinguish them from pure XVT-Power++ calls. In general, you do not need to learn the XVT Portability Toolkit to its fullest to use XVT-Power++.

3.2. A "Hello World" Program

As discussed in Chapter 1, any XVT-Power++ program must instantiate a `CApplication`-derived object, which can open one or more `CDocument`-derived objects. The document objects may then open one or more `CWindow`-derived objects to display some data. The "Hello World" program discussed here illustrates this process.

The goal of the program is to construct a window and display the friendly "`Hello World`" message as shown in Figure 2. Once this message displays, we will alter the text object so that it can change its message. The result will be an interactive windowing program that allows us to click on the text field and replace the "`Hello World`" message with another message that the user has typed. We will also be able to select a font from the Font and Style menus, and the text message will change accordingly.



Window when first invoked Window displaying a textual view

Figure 2. Shell Window

As is true for creating all XVT-Power++ programs, we do not start from scratch. Instead, we reuse three shell classes provided by XVT-Power++: `CShellApp`, `CShellDoc`, and `CShellWin`. These classes are derived from `CApplication`, `CDocument`, and `CWindow`, respectively. We take advantage of these shell classes throughout this tutorial because we can easily modify them to create applications, and they serve as examples of derived classes.

To develop the "Hello World" program, we must complete three general steps:

- Bring up a window
- Add a textual view to that window

- Then interact with the window

Before we begin these steps, we must make copies of the files for all three shell classes. We will work with these copies rather than with the original files. For some platforms, XVT-Power++ provides a simple utility program that creates a new shell project ready for you to compile and modify. For details on this utility program, see Section 7.9.

3.2.1. Bringing Up a Window

To bring up a window, we simply compile the shell application provided by XVT-Power++'s utility program. When we execute this program, a window like the first one shown in Figure 2 appears on the screen. As we work through this tutorial, you will become familiar with each of the shell classes and the flow of execution.

How you bring up a window depends upon your platform. XVT-Power++ supplies a shell application in the **demo/shell** directory. This application consists of the basic example shell application framework classes: `CShellApp`, `CShellDoc`, and `CShellWin`. The shell project provides a quick and easy way to get started. In most cases you should use these files when starting a new application rather than starting from scratch.

The next sections describe how to bring up a window on UNIX/Motif, Microsoft Windows, and Macintosh. You need to read only the section that is relevant to you.

3.2.1.1. UNIX/Motif

To bring up a window, simply compile the shell application provided by XVT-Power++'s utility program. When you execute this program, a window like the first one shown in Figure 2 appears on the screen. As you work through this tutorial, you will become familiar with each of the shell classes and the flow of execution.

Under the XVT-Power++ UNIX platform, you should use and reuse the shell project by using the `newdemo` utility provided in the **Pwr/util** directory. For example, to create a new XVT-Power++ application named *Hello*, enter the following:

```
newdemo Hello
make
```

`newdemo` will obtain fresh copies of the Shell project files and rename them using *Hello* as a base name. It also creates a makefile that will make the newly created *Hello* project.

3.2.1.2. Microsoft Windows 3.1

Under the XVT-Power++ Microsoft Windows platform, you should use and reuse the shell project by using the File Manager to make fresh duplicates of the entire **demo\shell** directory. The final XVT-Power++ release will provide an automated method of creating a new shell application with a user-defined name.

3.2.1.3. Macintosh

Under the XVT-Power++ Macintosh platform, you should use and reuse the shell project by means of the `NewPwrProject` script provided in the **Pwr:util** directory. For ease of use, add this script to your execution path (possibly by placing it in the **{MPW}Scripts** folder). For example, to create a new XVT-Power++ application named *Hello*, enter the following line at the MPW worksheet:

```
NewPwrProject Hello
```

`NewPwrProject` will obtain fresh copies of the Shell project files and rename them using *Hello* as a base name. It also creates a makefile that will make the newly created *Hello* application.

3.2.1.4. The Hello Project

Using XVT-Power++'s shell project on your platform, you should now have renamed files for `CHelloApp`, `CHelloDoc`, and `CHelloWin`. When you compile, a window appears on your screen.

3.2.2. Adding a View to the Window

Our goal now is to add a couple of views to the window we have created. The first view is a text message that says "`Hello World`"; the second is a text editing field into which the user can type any message. To accomplish this goal, we will use two of XVT-Power++'s view classes, `CText` and `NLineText`. First, we edit the `CHelloWin` class. We begin by modifying the class header, inserting the lines shown here in italics:

```
#include "CWindow.h"

class NLineText;
class CText;

class CHelloWin : public CWindow
{
    public:
        . . . code . . .
        . . . code . . .

    private:
        NLineText* itsField;
        CText* itsMessage;
};
```

Next, we modify the CHelloWin source file, as follows:

```
#include "PwrDef.h"
#include CText_i
#include NLineText_i

CHelloWin::CHelloWin(CDocument *theDocument, . . .
{
    itsMessage = new CText(this, CPoint(100,100),
            "Hello World");
    itsMessage->SetCommand(1);

    itsField = new NLineText(this, CPoint(100,200), 100);
    itsField->SetCommand(2);
}
```

At this point, we can compile our program and see that we have accomplished our goal.

Let's take a closer look at how the text object and text field were created. First, we added two private pointers for the views, itsField and itsMessage, to the CHelloWin class. Then we created the views in the constructor of the window:

```
itsmessage = new CText(this,CPoint(100,100),
    "Hello World");
```

new CText

This creates a new CText object, where CText is a class in the CView hierarchy. CViews are classes that, when instantiated, display themselves inside a window, may interact with the user, and automatically handle most events that come to them. The signature of the CText constructor is as follows:

```
CText(CSubview* theEnclosure,
    const CPoint& theTopLeft,
    const CString& theText = NULLString);
```

`this`
: The first parameter is the enclosure of the `CText` object. All views must be displayed inside an enclosure. `this` refers to the window being constructed.

`CPoint(100,100)`
: This is the `theTopLeft` parameter. It is a coordinate, relative to the `CText`'s enclosure, where the text will be displayed.

`"Hello World"`
: This is the `theText` parameter, which indicates what message should be displayed.

Similarly, we created the text field by instantiating the `NLineText` class. We also added two lines that set the command of each text object. A discussion of commands is deferred until the next section.

Notice how we included the headers for the `NLineText` and `CText` classes:

```
#include "PwrDef.h"
#include CText_i
#include NLineText_i
```

The **PwrDef.h** file contains definitions of `include` macros used to include XVT-Power++ class headers. These definitions allow us to include the class header for `CText` with the line `#include CText_i`. The use of this type of `include` idiom is explained in Chapter 2. This idiom allows us to use any XVT-Power++ class by obtaining its definition through a statement of the form `#include classname_i`.

Now we have a program that creates a window and displays the "`Hello World`" string inside it—as well as a text field. Notice that the "`Hello World`" message is an object created inside the window. This object has an identity, and it interacts with the rest of the application. This new object is automatically incorporated by XVT-Power++. It will receive events, repaint when necessary, and be deleted whenever its enclosure is destroyed. For example, try changing the font of the text by making a selection from the Font menu. XVT-Power++ automatically takes care of sending this message to the views and the window, which react by repainting themselves using the new font.

3.2.3. Interacting With the Window

Let's extend the "Hello World" program to explore some of the other behaviors and features that you can get out of a `CText` object. Specifically, we will change the program so that it can display any

string desired, rather than just "Hello World". A user will be able to enter any string into a text field and display it in the window.

Let's begin by considering how we can signal the CText object to change its message. Remember, CViews can interact with the user, and they respond to most events. XVT-Power++ has several channels of communication that allow different objects to send messages to each other. One such channel is the DoCommand method chain. All views generate a DoCommand message whenever they are clicked. Text fields also generate DoCommands when they receive a carriage return.

The DoCommand method of CHelloWin is called whenever one of the window's subviews generates a command message. There are specific rules that define the flow of DoCommand events in an XVT-Power++ application. For now, it suffices to know that the DoCommand reaches the window.

We open the CHelloWin source file and edit the DoCommand method, adding the following cases (shown in italics):

```
void CHelloWin::DoCommand(long theCommand, void* theData)
{
     switch (theCommand)
     {
          case 1:
               itsMessage->SetText("Hello World");
               DoDraw();
               break;

          case 2:
               itsMessage->SetText(itsField->GetText());
               DoDraw();
               break;

          default:
               CWindow::DoCommand(theCommand, theData);
     }
}
```

We used case 1 and case 2 to trap the commands sent by the different text objects because earlier in the program we assigned these commands as follows:

```
itsMessage->SetCommand(1);
itsField->SetCommand(2);
```

The first case statement takes care of command 1, which is generated by clicking within the text object. The two lines of code in this case are:

```
(1) itsMessage->SetText("Hello World!!!");
(2) DoDraw();
```

Line (1) tells `itsMessage` to set its text back to "`Hello World`". Line (2) calls the `CHelloWin::DoDraw` method in order to force an update of the window.

The second case statement takes care of command 2, which is generated by clicking in the edit field or entering a carriage return. The code in this case is very similar to that of case 1:

```
(1) itsMessage->SetText(itsField->GetText());
(2) DoDraw();
```

Line (1) tells `itsMessage` to reset its message to the text in `itsField`. We obtain the contents of the field by calling `itsField->GetText()`. Finally, we update the window by calling `DoDraw` in Line (2). Now, we can enter a message into the edit field, press the return key, and see the new message displayed in the window. If we click on the message, it reverts back to the original "`Hello World`" text.

3.2.4. Summary

The "Hello World" program is now complete. In just a few steps we have created an interactive windowing program. As you proceed with the tutorial, you will learn more about `CView` classes and how they work.

Here is a complete printout of the "Hello World" program. Since you did not modify the `CHellolApp` or `CHellolDoc` classes, they are omitted.

`CHelloWin.h`

```
#ifndef CHelloWin_HXX
#define CHelloWin_HXX

#include "CWindow.h"

class NLineText;
class CText;

class CHelloWin : public CWindow
{
    public:
    CHelloWin(CDocument *theDocument,
              const CRect& theRegion,
              const CString& theTitle = NULLString,
              long theWindowAttributes = WSF_NONE,
              WIN_TYPE theWindowType = W_DOC,
              int theMenuBarId = MENU_BAR_RID);

    BOOLEAN IHelloWin(BOOLEAN isBackgroundDrawn = TRUE,
```

```
            const CString& theTitle = NULLString,
            BOOLEAN isVisible = TRUE);

    virtual void DoCommand(long theCommand,void* theData=NULL);

    NLineText* itsField;
    CText* itsMessage;

};

#endif CHelloWin_HXX
```

CHelloWin.cxx

```
#include "PwrDef.h"

#include NLineText_i
#include CText_i

#ifdef STDH
#include "CHelloWin.h"
#endif STDH
#ifdef DOS
#include "CHelloWn.h"
#endif DOS
////////////////////////////////////////////////////////
CHelloWin::CHelloWin(CDocument *theDocument,
            const CRect& theRegion,
            const CString& theTitle,
            long theAttributes,
            WIN_TYPE theWindowType,
            int theMenuBar)

: CWindow(theDocument, theRegion, theTitle,
    theAttributes, theWindowType, theMenuBar)
{
    // Add any code to create sub-objects here.

    itsMessage = new CText(this, CPoint(100,100),
    "Hello World");
    itsMessage->SetCommand(1);

    itsField = new NLineText(this, CPoint(100,200), 100);
    itsField->SetCommand(2);
}
////////////////////////////////////////////////////////
BOOLEAN CHelloWin::IHelloWin(BOOLEAN isBackgroundDrawn,
            const CString& theTitle,
            BOOLEAN isVisible)
{
    // Add any code to initialize the window here.

    return
CWindow::IWindow(isBackgroundDrawn,theTitle,isVisible);
}
////////////////////////////////////////////////////////
```

```
void CHelloWin::DoCommand(long theCommand,void* theData)
{
    switch (theCommand)
    {
        case 1:
            itsMessage->SetText("Hello World");
            DoDraw();
            break;

        case 2:
            itsMessage->SetText(itsField->GetText());
            DoDraw();
            break;

        default:
            CWindow::DoCommand(theCommand,theData);
    }
}
```

3.3. A File Editing Program

In the preceding Hello World exercise, we modified the `CShellWin` class and worked with the `NLineText` and `CText` classes, which, like all `CView`-derived classes, can display themselves in a window and interact with the user. The exercise that follows gives us a chance to work with the `CShellDoc` class. This class is in charge of managing data and can display the data by instantiating a window and other `CView` objects. In developing the "Hello World" program, we saw how a single window is created. In developing the file editing program, we will explore how to create multiple documents, each of which can have multiple views.

In this exercise, we will develop a text file editor that can perform the following:

- Open a file for editing
- Allow the user to edit the file within a window
- Save the file back to disk
- Save and close before exiting the program.

We will begin with a menubar window that gives us access to a document window. Then we will display some data inside the window, as shown in Figure 3. To accomplish these tasks, we will modify all three shell classes.

Let's begin by making fresh copies of the files for `CShellApp`, `CShellDoc`, and `CShellWin` and renaming them `CEditApp`, `CEditDoc`, and `CEditWin`. Consult the Installationsection at the beginning of this *Guide* for information on how to do this.

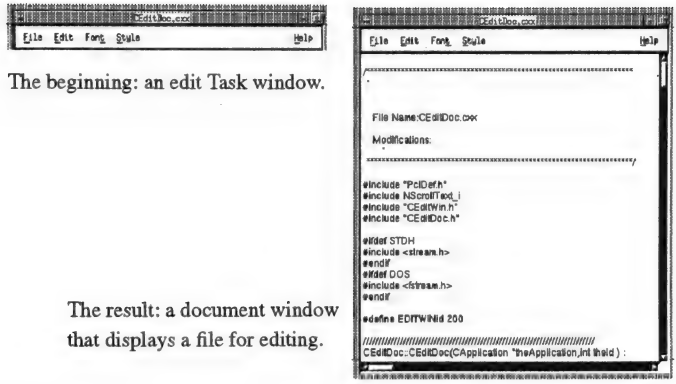


Figure 3. The Editing Windows

3.3.1. Opening a Document

In the "Hello World" program, we started by simply compiling the shell application and executing it. Right away, a window appeared on the screen. This happened because the shell application automatically opened a document as soon as it was executed. Our goal in developing the editing program is a little different. When the edit application is first launched, we want only the application's menubar to appear, allowing the user to open documents through the File menu. We can accomplish this goal by introducing the following changes to the shell Edit application.

First, we make fresh copies of the files for the shell classes—`CShellApp`, `CShellDoc`, and `CShellWin`—and rename them as `CEditApp`, `CEditDoc`, and `CEditWin`. XVT-Power++ supplies a shell application in the **demo/shell** directory.

Under the XVT-Power++ Macintosh platform, you can use and reuse the shell project by means of the `NewPwrProject` script provided in the **Pwr:util** directory. For ease of use, add this script to your execution path (possibly by placing it in the **{MPW}Scripts** folder). For example, to create a new XVT-Power++ application named *Edit*, enter the following line at the MPW worksheet:

```
NewPwrProject Edit
```

`NewPwrProject` will obtain fresh copies of the Shell project files and rename them using *Edit* as a base name.

Under the XVT-Power++ Microsoft Windows platform, you can use and reuse the shell project by means of the File Manager to make fresh duplicate copies of the entire **demo\shell** directory.

Under the XVT-Power++ UNIX platform, use the newdemo utility provided in the **Pwr/util** directory. For example, to create a new XVT-Power++ application named *Edit*, enter the following:

```
newdemo Edit
make
```

newdemo will obtain fresh copies of the Shell project files and rename them using *Edit* as a base name. It also creates a makefile that will make the newly created *Edit* project.

Using XVT-Power++'s shell project on your platform, you should now have renamed files for CEditApp, CEditDoc, and CEditWin. Now, we'll start by editing **CEditApp.h**

```
class CEditApp : public CApplication
{
public:

    ... code ...

    virtual void SetUpMenus(void);

    ... code ...
}
```

Next, we edit the CEditApp source file, modifying the StartUp method by removing the call to DoNew and adding the SetUpMenus method:

```
void CEditApp::StartUp(void)
{
    CApplication::StartUp();
}

void CEditApp::SetUpMenus(void)
{
    win_menu_enable(TASK_WIN, M_FILE_OPEN,TRUE);
    win_menu_enable(TASK_WIN, M_FILE_NEW, TRUE);
}
```

At this point we recompile and execute the Edit program. The Edit window that appears looks similar to the edit Task window shown in Figure 3, although its appearance differs from platform to platform. A user can select "Open" or "New" from the File menu and bring up a window. If the user selects "Open", a file dialog box appears that queries the user for the name of the file to open (see Figure 4).

If the window is closed, we can select "Open" or "New" again and bring up a new window. The "Exit" item on the File menu causes the application to terminate as expected.

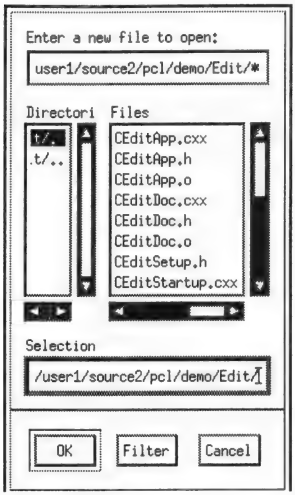


Figure 4. File Dialog Box for an "Open" Operation

Let's take a closer look at how we accomplished this behavior in the Edit program. XVT-Power++ calls the `StartUp` method of `CEditApp` to indicate that the application can begin any startup procedures. Before we edited the `StartUp` method, `CEditApp` responded to `StartUp` events by calling `DoNew` to create a new document. This is why a window is created automatically as soon as the "Hello World" program executes. Removing the call to `DoNew` ensures that the document and any of its windows will not be created right away. Actually, since our edited `StartUp` method now simply calls the inherited method, we can remove the entire method from the `CEditApp` class.

Also, we added a method called `SetUpMenus`. This method is a virtual `CApplication` method that XVT-Power++ calls to set up the initial application menubar. Inside the method, we placed two lines of XVT Portability Toolkit code to enable the "Open" and "New" items on the File menu.

```
win_menu_enable(TASK_WIN, M_FILE_OPEN,TRUE);
win_menu_enable(TASK_WIN, M_FILE_NEW, TRUE);
```

XVT-Power++ handles menubar events by sending a `DoMenuCommand` message to the appropriate object. In our case, if "Open" is selected from the menubar, the menu command goes

directly to the CEditApp object. Internally, however, XVT-Power++ recognizes the "Open" menu item as special and translates it into a call to CEditApp's DoOpen method. If you look into the CEditApp code, you will see the following lines:

```
BOOLEAN CEditApp::DoOpen(void)
{
    CEditDoc* aDoc=new CEditDoc(this,GetNumDocuments()
        +1);
    PwrAssert(aDoc, 311, "Creation of CEditDoc failed");
    return aDoc->DoOpen();
}
```

Line (1) instantiates the CEditDoc. Line (2) asserts that the creation was successful. Finally, line (3) returns the value obtained from calling the new document's DoOpen method. By default, CDocument::DoOpen brings up the file dialog box, if necessary, and builds a new window. In summary, the user selects the "Open" menu item from the File menu. This selection is translated into a call to the application object's DoOpen method, which in turn creates a new document and passes the DoOpen message to it.

3.3.2. Viewing the Document's Data

Now that we can open documents and document windows, we are ready to display the data. For the Edit application, this involves displaying the contents of a file inside a window like the one shown in Figure 3.

First, we edit the CEditDoc header file:

```
class NScrollText;

class CEditDoc : public CDocument<method>
{
    public:
        . . . code . . .
        . . . code . . .

    private:
        NScrollText* itsText;
};
```

Next, we edit the CEditDoc source file, adding the following includes and define line and editing the BuildWindow method:

```
#include "PwrDef.h"
#include NScrollText_i
#include <stream.h>
    //this may actually be <fstream.h> in some systems
```

```
#define EDITWINid 200

     . . . code . . .

void CEditDoc::BuildWindow(void)
{
    // Build an instance of CEditWin:

    // First, build the window:
    CEditWin* aWin = new CEditWin(this,
        CRect(20,30,420,430), "Edit Window",
        WSF_CLOSE|WSF_SIZE|WSF_ICONIZABLE|
        WSF_HSCROLL|WSF_VSCROLL);
    aWin->SetId(EDITWINid);

    // Now build the scroll-text and place it inside the
    // window:
    itsText = new NScrollText(win);

    // Finally, open the file and pipe the text into the
    // scroll-text:
    if (itsXVTFilePointer)
    {
        aWin->SetTitle(itsXVTFilePointer->name);

        CString fileText;
        ifstream fileStream(itsXVTFilePointer->name);

        itsText->Suspend();

        if (fileStream)
        while (!fileStream.eof())
        {
            fileText.Clear();
            fileStream >> fileText; // Read until '\n'
            itsText->AddParagraph(fileText, USHRT_MAX);
        }

        itsText->Resume();
    }
}
```

When we recompile the program, we are able to open a file and display it inside a window. We can edit the text in the window, scroll it, and change its font.

Let's consider the specifics of how `BuildWindow` works. `BuildWindow` is a `CDocument` pure virtual method that must be defined by all `CDocument`-derived classes. This method is automatically called from within `CDocument`'s `DoOpen` and `DoNew` methods. Inside `BuildWindow`, you should define the specifics for building the document's views. `BuildWindow` first creates an instance of `CEditWin` and sets the window's ID:

```
CEditWin* aWin = new CEditWin(this, CRect(20,30,420,430),
    "Edit Window", WSF_CLOSE|WSF_SIZE|WSF_ICONIZABLE|
    WSF_HSCROLL|WSF_VSCROLL)
    win->SetId(EDITWINid);
```

To the `CEditWin` constructor, we supplied information on its document, its coordinates, title, and other general window attributes. Then we set the window's ID using the `EDITWINId` define.

The next line create a new scroll text object:

```
itsText = new NScrollText(win);
```

`NScrollText` is a class that provides a text editing box with scrollbars. You may want to look up the constructor and the initializer in the *XVT-Power++ Reference* to find out the meaning of each parameter.

Finally, `BuildWindow` opens a file and pipes its contents into the scroll text.

```
CString fileText;
ifstream fileStream(itsXVTFilePointer->name);

itsText->Suspend();

if (fileStream)
    while (!fileStream.eof())
    {
        fileText.Clear();
        fileStream >> fileText;
        itsText->AddParagraph(fileText, USHRT_MAX);
    }
itsText->Resume();
```

As you can see, an input file stream is created, and the contents of the file are extracted into a `CString` object, which in turn is added into the `NScrollText` object. Notice that this code is wrapped between calls to the `NScrollText Suspend` and `Resume` methods. This wrapping suspends any redrawing of the text until all paragraphs have been added.

In the `BuildWindow` method, we also use a protected `CDocument` structure provided by `itsXVTFilePointer`. This structure is used to store the document's filename and is set automatically when the user selects a name from the filename dialog box.

3.3.3. Saving the Data

Now we can edit a file to as much as we want to, but all of our work will be lost as soon as we exit the application or close the edit window. We still need to add code that saves a file. Also, we must specify how the user will be able to save the data. When a document is first opened and the user has made no changes, there is no need to save anything. However, as soon as the user modifies the view by

typing into the edit window, the document must recognize that the data needs to be saved, and the "Save" item on the File menu must be enabled.

In addition, if the user attempts to close the edit window or exit the application without first saving any changes made, the program should display a dialog box (like the one in Figure 5) that gives the user a chance to save before exiting. Finally, if the document has no associated filename or if the user selects "Save As", then a "Save As" dialog box, as shown in Figure 5, should appear.

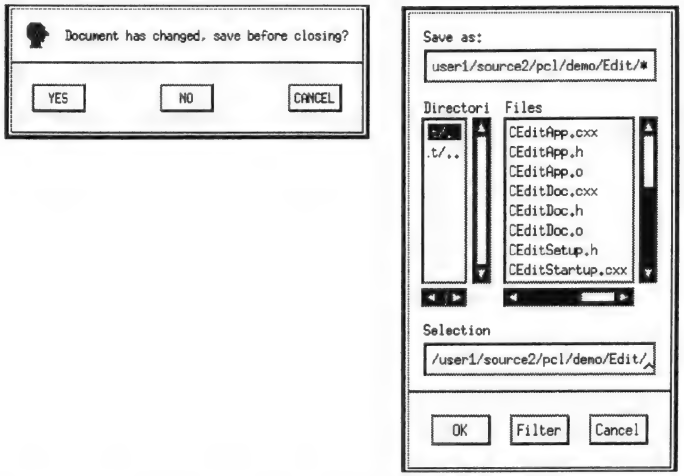


Figure 5. The "Save" and "Save As" Dialog Boxes

We begin by adding the following methods to the **CEditDoc header** file:

```
class CEditDoc : public CDocument
{
    public:
        . . . code . . .
        . . . code . . .

        virtual BOOLEAN DoSave(void);
        virtual BOOLEAN DoSaveAs(void);
        virtual void UpdateMenus(void);

        . . . code . . .
};
```

Next, we define these methods in the `CEditDoc` source file as follows:

```
BOOLEAN CEditDoc::DoSave()
{
    // Handle File menu Save

    if (!CDocument::DoSave())
        return FALSE;

    // Save the file:
    if (itsText)
    {
        ofstream fileStream(itsXVTFilePointer->name);
        fileStream << itsText->GetText();
        fileStream.close();
    }

    return TRUE;
}

BOOLEAN CEditDoc::DoSaveAs(void)
{
    // Override CDocument::DoSaveAs to set the
    // window's title.

    if (CDocument::DoSaveAs())
    {
        CWindow* window = FindWindow(EDITWINid);
        if (window)
            window->SetTitle(itsXVTFilePointer->name);
        return TRUE;
    }
    else
        return FALSE;
}

void CEditDoc::UpdateMenus(void)
{
    // Set the Save menu item according to the needs of the
document:

    CWindow* window = FindWindow(EDITWINid);
    if (!window) return;

    if (itsXVTFilePointer)

win_menu_enable(window->GetXVTWindow(),M_FILE_SAVE,NeedsSaving()
);

    win_menu_enable(window->GetXVTWindow(),M_FILE_SAVE_AS,
NeedsSaving());
}
```

We also add a method to the `CEditWin` header file, as follows:

```
class CEditWin : public CWindow
{
    public:
        . . . code . . .
        . . . code . . .

    virtual void Key(int ch, BOOLEAN isShiftKey,
            BOOLEAN control);
        . . . code . . .
};
```

We define the Key method in the CEditWin source file:

```
#include "PwrDef.h"
#include CDocument_i

void CEditWin::Key(int /*ch*/,BOOLEAN /*isShiftKey*
    ,BOOLEAN /*control*/)
{
    // Trap keyboard events to set the document
    //  needSave flag.

    itsDocument->SetSave(TRUE);
}
```

Now, when we compile and execute the Edit program, all of the features that we set as our goal are incorporated into the program. You can edit files, save them to disk, and open them again to verify that the editor actually works.

Let's consider in detail how we achieved the goals set for the Edit program. We first added the definition of DoSave to the document. DoSave is a CDocument virtual method that is called automatically when the user selects "Save" from the File menu. In our implementation of DoSave, we call the inherited DoSave method, which takes care of bringing up a file dialog box if no filename has been set for this document. DoSave verifies that itsText is not NULL and saves the document by opening itsFile for output and inserting the text extracted by the call to the NScrollText GetText method.

Similarly, we defined DoSaveAs, a CDocument virtual method that is called when the user selects "Save As" from the File menu. We call the inherited DoSaveAs method to bring up the appropriate filename dialog box. If this call is successful, we then proceed to set the window's title, as follows:

```
CWindow* window = FindWindow(EDITWINid);
if (window)
    window->SetTitle(itsXVTFilePointer->name);
```

Notice that we call FindWindow to get a handle on the document's edit window. FindWindow is a CDocument method that can find a document window, given the ID of that window. Recall that inside BuildWindow, we set the window's ID as follows:

```
win->SetId(EDITWINid);
```

When you execute the program, the "Save" and "Save As" File menu items are enabled or disabled, depending on whether the document actually needs to be saved. Each document contains a private flag that indicates whether its data needs saving. This flag is

accessible through the NeedsSaving and SetSave methods. Notice how we use NeedSaving in the definition of UpdateMenus:

```
if (itsXVTFilePointer)
    win_menu_enable(window->GetXVTWindow(),M_FILE_SAVE,
        NeedsSaving());

win_menu_enable(window->GetXVTWindow(),M_FILE_SAVE_AS,
    NeedsSaving());
```

The first line enables or disables the "Save" menu item only if the document already has an assigned filename. The "Save As" menu item, however, is set by the NeedSaving flag, regardless of whether the document needs to be saved. The state of NeedSaving is automatically updated by XVT-Power++ when the document's DoSave or DoSaveAs methods are called. In addition, changing this state triggers a call to the document's UpdateMenus virtual method, which we have just defined.

Finally, we added a Key method to the window. Remember, we want to ensure that the document's NeedSaving flag is set if the user ever modifies the view by typing into the window. When the user does type into the window, CWindow's virtual Key method is automatically called. Therefore, we overrode Key in the CEditWin class and made the following call:

```
itsDocument->SetSave(TRUE);
```

3.3.4. Summary

We have completed our second XVT-Power++ application. In a short time, we have implemented a simple text editor. Here is the complete listing of the Edit Program:

CEditApp.h

```
#ifndef CEditApp_H
#define CEditApp_H

#include "PwrDef.h"
#include CApplication_i

class CEditApp : public CApplication
{
    public:
        CEditApp(void);

        virtual BOOLEAN DoNew(void);
        virtual BOOLEAN DoOpen(void);
        virtual void SetUpMenus(void);
};

#endif CEditApp_H
```

CEditApp.cxx

```
#include "CEditApp.h"
#include "CEditDoc.h"
#include "xvtmenu.h"

////////////////////////////////////////////////////////
CEditApp::CEditApp(void) : CApplication()
{

}
////////////////////////////////////////////////////////
BOOLEAN CEditApp::DoNew(void)
{
     CEditDoc* aDoc = new CEditDoc(this,
          GetNumDocuments()+1);
     PwrAssert(aDoc, 311, "Creation of CEditDoc failed");
     return aDoc->DoNew();
}
////////////////////////////////////////////////////////
BOOLEAN CEditApp::DoOpen(void)
{
     CEditDoc* aDoc = new CEditDoc(this,
          GetNumDocuments()+1);
     PwrAssert(aDoc, 311, "Creation of CEditDoc failed");
     return aDoc->DoOpen();
}
////////////////////////////////////////////////////////
void CEditApp::SetUpMenus(void)
{
     win_menu_enable(TASK_WIN, M_FILE_OPEN,TRUE);
     win_menu_enable(TASK_WIN, M_FILE_NEW, TRUE);
}
```

CEditDoc.h

```
#ifndef CEditDoc_HXX
#define CEditDoc_HXX

#include "PwrDef.h"
#include CDocument_i

class NScrollText;

class CEditDoc : public CDocument
{
     public:
          CEditDoc(CApplication *theApplication,int theId );
          virtual void BuildWindow(void);

          virtual BOOLEAN DoSave(void);
          virtual BOOLEAN DoSaveAs(void);

          virtual void UpdateMenus(void);

     private:

          NScrollText* itsText;

 };

#endif CEditDoc_HXX
```

CEditDoc.cxx

```
#include "PwrDef.h"
#include NScrollText_i
#include "CEditWin.h"
#include "CEditDoc.h"

#ifdef STDH
#include <stream.h>
#endif
#ifdef DOS
#include <fstream.h>
#endif

#define EDITWINid 200

////////////////////////////////////////////////////
CEditDoc::CEditDoc(CApplication *theApplication,
    int theId ) :
            CDocument(theApplication,theId)
{
    itsText = NULL;
}
////////////////////////////////////////////////////
void CEditDoc::BuildWindow(void)
{
    // Build an instance of CEditWin:

    // First, build the window:
    CEditWin* win = new CEditWin(this, CRect(20, 30, 420, 430), "untitled",
WSF_CLOSE|WSF_SIZE|WSF_ICONIZABLE|WSF_HSCROLL|WSF_VSCROLL);

    win->SetId(EDITWINid);

    // Now build the scroll-text and place it inside the window:
    itsText = new NScrollText(win);

    // Finally, open the file and pipe the text into the scroll-text:
    if (itsXVTFilePointer)
    {
        win->SetTitle(itsXVTFilePointer->name);

        CString fileText;
        ifstream fileStream(itsXVTFilePointer->name);

        itsText->Suspend();

        if (fileStream)
            while (!fileStream.eof())
            {
                fileText.Clear();
                fileStream >> fileText;  // Read until '\n'
                itsText->AddParagraph(fileText, USHRT_MAX); // Add to end of
                    //text
            }

        itsText->Resume();
    }
}
////////////////////////////////////////////////////
BOOLEAN CEditDoc::DoSave()
{
    // Handle File menu Save
```

```
    if (!CDocument::DoSave())
        return FALSE;

    // Save the file:
    if (itsText)
    {
        ofstream fileStream(itsXVTFilePointer->name);
        fileStream << itsText->GetText();
        fileStream.close();
    }

    return TRUE;
}
////////////////////////////////////////////////////////
BOOLEAN CEditDoc::DoSaveAs(void)
{
    // Override CDocument::DoSaveAs to set the windows title.

    if (CDocument::DoSaveAs())
    {
        CWindow* window = FindWindow(EDITWINid);
        if (window)
            window->SetTitle(itsXVTFilePointer->name);
        return TRUE;
    }
    else
        return FALSE;
}
////////////////////////////////////////////////////////
void CEditDoc::UpdateMenus(void)
{
    // Set the Save menu item according the the needs of the document:

    CWindow* window = FindWindow(EDITWINid);
    if (!window) return;

    if (itsXVTFilePointer)
    win_menu_enable(window->GetXVTWindow(), M_FILE_SAVE, NeedsSaving());

    win_menu_enable(window->GetXVTWindow(), M_FILE_SAVE_AS,
            NeedsSaving());
}
```

CEditWin.h

```
#ifndef CEditWin_HXX
#define CEditWin_HXX

#include "CWindow.h"

class CEditWin : public CWindow
{
    public:

        CEditWin(CDocument *theDocument,
            const CRect& theRegion,
            const CString& theTitle = NULLString,
            long theWindowAttributes = WSF_NONE,
            WIN_TYPE theWindowType = W_DOC,
            int theMenuBarId = MENU_BAR_RID);
```

```
        virtual void Key(int ch, BOOLEAN isShiftKey, BOOLEAN control);
        virtual BOOLEAN Close(void);
};
#endif CEditWin_HXX
```

CEditWin.cxx

```
#include "PwrDef.h"
#include CDocument_i

#include "CEditWin.h"

//////////////////////////////////////////////////////
CEditWin::CEditWin(CDocument *theDocument,
            const CRect& theRegion,
            const CString& theTitle,
            long theAttributes,
            WIN_TYPE theWindowType,
            int theMenuBar)

: CWindow(theDocument, theRegion, theTitle,
            theAttributes, theWindowType, theMenuBar)
{
    // Add any code to create sub-objects here.
}
//////////////////////////////////////////////////////
void CEditWin::Key(int /*ch*/,BOOLEAN /*isShiftKey*/,BOOLEAN /*control*/)
{
    // Trap keyboard events to set the document needSave flag.

    itsDocument->SetSave(TRUE);
}
//////////////////////////////////////////////////////
BOOLEAN CEditWin::Close(void)
{
    // Give the user a chance to save if needed before closing window.

    if (itsDocument->NeedsSaving())
    {
        switch (xvt_ask("YES","NO","CANCEL","File has changed, save?"))
        {
            case RESP_DEFAULT: itsDocument->DoSave();
                return CWindow::Close();

            case RESP_2:  return CWindow::Close();

            case RESP_3:  return FALSE;
        }
    }
    else
        return CWindow::Close();
}
```

3.4. A Drawing Program

This section of the tutorial guides you through the process of building yet another application—a drawing program. The emphasis now is on learning the structure of an *entire* XVT-Power++ application while examining such specific features of the library as

data propagation, menubar command handling, resource creation, view nesting, view manipulation, and scrolling.

To illustrate resource creation, the Draw program, when launched, will display a dialog box, called the About box, giving information about the Draw application (see Figure 8).

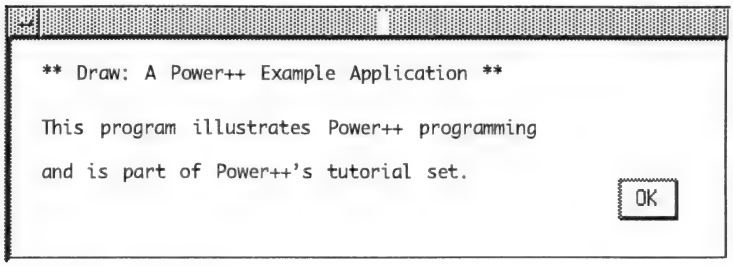


Figure 6. About Box

The drawing program will consist of a window that permits users to create drawable objects dynamically. These objects will be sizable, draggable, and scrollable. In addition, it will display a textual representation of the drawing window that will track the user's actions by displaying the type of objects created and their location. See Figure 7. Also, if an object is sized or dragged, the action is represented in the text window as new coordinates for the object. The textual representation can be saved into a file before the document is closed because the drawing program incorporates all of the file management features of the editing program discussed in the preceding section of this tutorial. Finally, only one document at a time can be open in the drawing program.

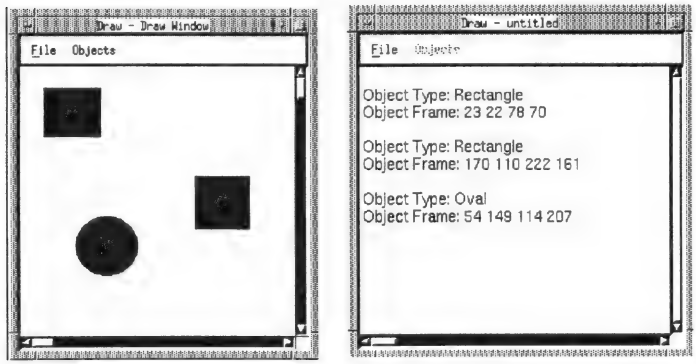


Figure 7. Drawing Window and Its Textual Representation

3.4.1. Starting a Program

Until now, you have not seen how an XVT-Power++ program actually begins executing. Execution occurs inside the `main()` function of your application. You can find `main()` inside the `CShellStartup` source file, now renamed **CDrawStartup.cxx**. The file contains the following code:

```
#include "CDrawApp.h"

extern "C"
{
    void main(int argc, char* argv[])
    {
    CDrawApp theApplication;
    theApplication.Go(argc,argv,MENU_BAR_RID,0,"Draw",
        "Draw", "Draw");
    }
}
```

In all XVT-Power++ programs, you must define a `main` function, which creates an instance of your `CApplication`-derived class and invokes the `Go` method. Note that the function has been wrapped into an extern "C" block. This is not always needed, but under some compilers your program will fail to link unless `main` has C linkage.

Once you invoke `Go`, control is transferred to XVT-Power++ and never returned, so no code should follow `Go`. Very little—if anything else—is done inside `main`.

See Also: For a full discussion of the role of `main`, consult Chapter 4 in this *Guide* and the `CApplication` section in the *XVT-Power++ Reference*.

Following is the definition of `Go`. Compare it with the actual use of `Go` in the preceding lines from the `CDrawStartup` source file.

```
void CApplication::Go(int argc, char *argv[],
                      short theMenuBarId,
                      short theAboutBoxId,
                      char *theBaseName,
                      char *theApplicationName,
                      char *theTaskTitle)
```

See Also: For a full description of each parameter, consult the `CApplication` section in the *XVT-Power++Reference.*

In the `CDrawStartup` source file, the following values have been supplied to each parameter:

`arcg, argv`
: These parameters passed literally as they were supplied to `main`.

`theMenuBarId`
: We passed in `MENU_BAR_RID`, which corresponds to a standard menubar as long as no other menubars are specified. We will soon tell you how to change this value.

`theAboutBoxId`
: We passed in a value of `0`, which designates the standard XVT Portability Toolkit About box. We will soon tell you how to change this value as well.

`theBaseName`
: We passed in "`Draw`" because this is the name of the application stored on disk.

`theApplicationName`
: The string passed to this parameter is used as a title for the windows belonging to this application. We supply "Draw", but any other name will do.

`theTaskTitle`
: The string passed in is used as the title of the application's task window. Task windows are visible only under certain platforms (Windows and Presentation Manager).

See Also: For more information, see the section on `CTaskWin` in the *XVT-Power++ Reference*.

3.4.2. Defining the Resources for the Program

Very often, XVT-Power++ programs use graphical and textual resources that are defined and compiled separately from the program and loaded at run time. Resources such as menubars, dialog boxes, strings, and so on, are coded using the XVT Portability Toolkit's Universal Resource Language (URL). Teaching you to use URL is beyond the scope of this presentation, but the *XVT Guide* includes information on using URL. This section shows you how the resources for the Draw program are defined and created.

We will create the following three types of resources:

- A dialog box that is used as the About box of the application
- A menubar that contains menus specific to the Draw application (as appear on the windows in Figure 7)
- Two string resources

We begin by creating a new file called **Draw.h** in which to define macros (using `#define`) to use throughout the Draw files:

```
/*******************************************************

File Name: Draw.h

This file contains global definitions for the tutorial's Draw app.

*******************************************************/

#define ABOUTBOXRid    100 /* Resource id for the about box */
#define ABOUTText1     101 /* Resource ids for the about box's text ... */
#define ABOUTText2     102
#define ABOUTText3     103

#define MENUBarRid     104 /* Resource id for the application menubar */
#define OBJECTMenu     105 /* Id for the "Objects" menu */
#define RECTANGLEid    106 /* Menu item Id for "Rectangle" object */
#define OVALid         107 /* Menu item Id for "Oval" object */
                           /* CView* is passed as theData. */

#define SKETCHcmd      108 /*Command generated by the sketchpad. */
#define NEWObjectCmd   109 /*Generated when a new view is created.*/
```

Next, we define the application's resources inside the **Draw.url** file.

```
#define APPNAME draw
#define QAPPNAME "Draw"

#include "url.h"
#include "PwrURL.h"

#include "Draw.h"

/* Definition of the about box dialog box: */
DIALOG ABOUTBOXRid 100,100,390,120
     BUTTON DLG_CANCEL, 330, 75, 35, 25 "OK" DEFAULT
     TEXT ABOUTText1 15, 15, 360, 15 "** Draw: A Power++ Example Application **"
     TEXT ABOUTText2 15, 40, 360, 15 "Program illustrates Power++ programming"
     TEXT ABOUTText3 15, 60, 360, 15 "and is part of Power++'s tutorial set."

/* Define the application menubar: */

MENUBAR MENUBarRid

MENU MENUBarRid
     DEFAULT_FILE_MENU
     SUBMENU OBJECTMenu "Objects" DISABLED

MENU OBJECTMenu "Objects"
     ITEM RECTANGLEid "Rectangle" CHECKED
     ITEM OVALid "Oval" CHECKABLE

/* Define string resources used for object type names: */

STRING RECTANGLEid "Rectangle"
STRING OVALid "Oval"
```

At this point, you do not have to recompile your program in order to incorporate these new resources. However, you do have to recompile the **draw.url** file. Consult the Installation section at the beginning of this *Guide* to become acquainted with how resources are compiled for your platform.

Let's look at the specifics of how the different resources are defined. First, we define a dialog box with a resource ID of `ABOUTBOXid`. This dialog box has a top-left screen coordinate of (100,100) and a size of 390 by 120 pixels. Finally, it has a button and three static text controls. Figure 8 shows the dialog box as it looks under a particular platform.

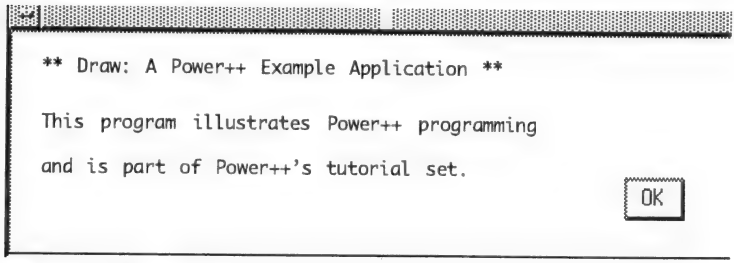


Figure 8. Dialog Box

XVT-Power++ allows us to define a different menubar for each window in an application. Here, we defined one menubar that we will use throughout. The new `MENUBAR` has a `MENUBarRid` as a resource ID. It has two menus: the default *File* menu provided by the XVT Portability Toolkit and an *Objects* menu that you have defined. The Objects menu contains two choices of objects that a user can draw: "Rectangle" and "Oval". Later, we could add more choices to this menu. Notice that the Objects menu is initially disabled and that both items in this menu are checkable. Also notice that each of the menu items has an associated ID, as shown here in boldface type:

```
ITEM RECTANGLEid "Rectangle" CHECKED
ITEM OVALid "Oval" CHECKABLE
```

XVT-Power++ uses this ID as a command number for sending events that are generated when the user selects one of these items from the Objects menu.

Finally, we define two string resources, one for rectangles and one for ovals. The ID for each of these resources is the command number given to each of the items on the Objects menu. The relationship between the ID number of the resource and the command number of the menu item is crucial to the Draw program, as we shall see later in this discussion.

3.4.3. Using the Program Resources

In the preceding section, we defined the resources to be used by the Draw program; here, we'll see how the resources are actually used within the application. Specifically, we will display the About box as the application is launched and use the menubar in the

application's windows. We begin by modifying the call to Go inside of main.

```
#include "CDrawApp.h"
#include "Draw.h"

extern "C"
{
    void main(int argc, char* argv[])
    {
        CDrawApp theApplication;
        theApplication.Go(argc, argv, MENUBarRid,
            ABOUTBOXRid, "Draw", "Draw",  "Draw");
    }
}
```

To display the About box as soon as the application is launched, we change the CDrawApp::CStartup method, as follows:

```
void CDrawApp::StartUp(void)
{
    CApplication::StartUp();
    about_box();
}
```

Finally, we change the CDrawDoc::BuildWindow method:

```
void CDrawDoc::BuildWindow(void)
{
    // Build an instance of CDrawWin:

    CDrawWin* aWin = new CDrawWin(this,
        CRect(20,20,400,400), "Draw
        Window", WSF_SIZE|WSF_ICONIZABLE, W_DOC,
        MENUBarRid);
}
```

In order to create the About box and menubar, we simply modified the call to Go, initializing the application using the resources defined in **Draw.url**. Now, when we compile and execute the Draw program, we see the newly created About box as well as the Draw-specific menubar.

Remember, StartUp is a virtual method that is called as soon as XVT-Power++ has finished initializing and is ready to start processing events. While developing the Edit program, we removed the call to DoNew. Now, we replace that call with a call to about_box, an XVT function that creates the application's About box as supplied to the Go method. Finally, we modified the document's BuildWindow method so that the window created is initialized with the correct menubar.

3.4.4. Defining Documents for the Program

In this section, we will define the behavior of the Draw document as appropriate for the Draw program. We will allow the user to create new documents, each of which will display two views of its data. We also will ensure that only one document can be opened at one time.

As in the Edit program, we start by adding a `SetUpMenus` method to our application class:

```
class CDrawApp : public CApplication
{
public:

    ... code ...

    virtual void SetUpMenus(void);

    ... code ...

};

void CDrawApp::SetUpMenus(void)
{
     win_menu_enable(TASK_WIN, M_FILE_NEW, TRUE);

}
```

The next step is to edit the CDrawApp::DoNew method as follows:

```
BOOLEAN CDrawApp::DoNew(void)
{
    if (DoClose())
    {
        CDrawDoc* aDoc = new CDrawDoc(this,GetNumDocuments()+1);
        PwrAssert(aDoc, 311, "Creation of CDrawDoc failed");
        return aDoc->DoNew();
    }
}
```

We then modify the document by adding `UpdateMenus` and `DoSaveMenu` methods and by modifying the `BuildWindow` method. All of the code for these steps is shown here and is very similar to the code we added for the document in the Edit program.

We modify the document by first editing the **CDrawDoc** header file, as follows:

```
class CWindow;
class CDrawWin;
class NScrollText;

class CDrawDoc : public CDocument
{
    public:

        ... code ...
        virtual void UpdateMenus(void);
        virtual BOOLEAN DoSave(void);
```

```
    private:

        CDrawWin* itsDrawWin;
        CWindow* itsTextWin;
        NScrollText* itsText;
};
```

In the **CDrawDoc** source file, we add the appropriate includes, modify the constructor, and define the appropriate methods:

```
#include "PwrDef.h"
#include NScrollText_i
#include CWindow_i

#include "Draw.h"

#include <stream.h>        // may need to be fstream.h

CDrawDoc::CDrawDoc(CApplication *theApplication, int theId) :
                CDocument(theApplication, theId),
                itsDrawWin(NULL), itsTextWin(NULL), itsText(NULL)
{
}

void CDrawDoc::BuildWindow(void)
{
    PwrAssert((itsDrawWin==NULL) && (itsTextWin==NULL),1,
            "BuildWindow() called with existing windows");

    // First build a graphical view:
    itsDrawWin = new CDrawWin(this, CRect(20,20,300,300), "Draw Window",
                WSF_SIZE|WSF_ICONIZABLE|WSF_HSCROLL
                |WSF_VSCROLL, W_DOC, MENUBarRid);

    // Next build a textual view:
    itsTextWin = new CWindow(this, CRect(120,20,420,280), "Log Window",
                WSF_SIZE|WSF_ICONIZABLE|WSF_HSCROLL|
                    WSF_VSCROLL);

    itsText = new NScrollText(itsTextWin, TX_BORDER|TX_READONLY
                    |TX_AUTOVSCROLL);
}

void CDrawDoc::UpdateMenus(void)
{
    CWindow* win;
    CIterator doTo(itsWindows);
    while (win = (CWindow*) doTo.Next())
    {
        win_menu_enable(win->GetXVTWindow(),M_FILE_SAVE_AS, NeedsSaving());

        if (itsXVTFilePointer)
         win_menu_enable(win->GetXVTWindow(), M_FILE_SAVE, NeedsSaving());
    }
}

BOOLEAN CDrawDoc::DoSave(void)
{
    // Handle File menu Save
    if (!CDocument::DoSave())
        return FALSE;
```

```
    // Save the file:
    if (itsText)
    {
        ofstream fileStream(itsXVTFilePointer->name);
        fileStream << itsText->GetText();
        fileStream.close();
    }

    itsDrawWin->SetTitle(itsXVTFilePointer->name);
    if (itsTextWin)
        itsTextWin->SetTitle(itsXVTFilePointer->name);

    return TRUE;
}
```

After we recompile and execute the program, we can open documents, and each time we do, two windows are created. When we select "New" from the File menu, the currently open document is closed and a new one is created. Let's explore the details of how we achieved our goals in this phase of the Draw program's development.

We began by editing the application, overriding `SetUpMenus` to enable the "New" item on the File menu. We also modified `CDrawApp::DoNew()` so that it would call `DoClose` before opening a new document, ensuring that only one document is open at one time. In addition we made several changes to the `CDrawDoc` class. We redefined `BuildWindow` to create two windows. First, we created the graphical window:

```
PwrAssert((itsDrawWin==NULL) && (itsTextWin==NULL),1,
            "BuildWindow() called with existing windows");

    // First build a graphical view:
    itsDrawWin = new CDrawWin(this, CRect(20,20,300,300), "Draw Window",
                 WSF_SIZE|WSF_ICONIZABLE, W_DOC, MENUBarRid);
```

`PwrAssert` ensures that `BuildWindow` is not called twice. The rest of the method simply creates the text window and the scroll text in a way similar to what you have seen so far. The textual view is almost identical to the one used in the Edit example.

Next, we defined the `DoSave` and `UpdateMenus` methods used to save the document to disk. As in the Edit program, `DoSave` opens a file and dumps in the contents of the scroll text. It also updates the window's titles to match the filename.

`UpdateMenus` enables/disables the "Save" and "Save As" menu items as appropriate, depending on the value of `NeedsSaving` and the existence of a filename. It iterates through all the document's windows and updates the menus. We could simply update the menus directly using the pointers `itsDrawWin` and `itsTextWin`. We have chosen to iterate though the document's list of windows

(itsWindows) in order to illustrate two points. First, all documents have a list of windows called itsWindows. Second, we can iterate through the list using a list iterator.

See Also: For more information on lists and iterators, consult the sections on CList and CIterator in the *XVT-Power++Reference*.

3.4.5. Setting Up a Drawing Window

Now we continue development by creating a scroller (using the CScroller class) inside the graphical window and inserting into it a sketchpad that creates graphical objects (ovals and rectangles) as the user drags the mouse on the sketchpad. Basically, an internal window data member keeps track of the type of object currently being created. The user selects the type of the object through the Objects menu on the menubar. CScroller is a class that enables us to scroll views inside a window. We create a scroller and then use it as the enclosure for any view inside its boundaries. XVT-Power++ takes care of all scrolling from here on.

Our first step in creating a scrollable sketchpad is to edit the CDrawWin source file as follows:

```
#include "PwrDef.h"
#include CScroller_i
#include CSketchPad_i
#include CRectangle_i
#include COval_i
#include CDocument_i

#include "CDrawWin.h"
#include "Draw.h"

CDrawWin::CDrawWin(CDocument *theDocument, ...
{
    // Create the scroller:
    itsScroller = new CScroller(this, 1000, 1000);
    itsScroller->IScroller(TRUE, TRUE, TRUE, 10, 50);
    itsScroller->SetGlue(ALLSTICKY);

    // Create a sketchpad inside the scroller:
    itsSketchPad = new CSketchPad(itsScroller,
                   itsScroller->GetVirtualFrame());
    itsSketchPad->SetCommand(SKETCHcmd);
    itsSketchPad->SetSketchEverywhere(FALSE);
}
```

We add the data members itsScroller and itsSketchPad to CDrawWin's definition:

```
class CScroller;
class CSketchPad;
```

```
class CDrawWin : public CWindow
{
        ... code ...

    private:

        CScroller* itsScroller;
        CSketchPad* itsSketchPad;
};
```

When we recompile and execute the program, you can drag the mouse across the drawing window, sketching rectangular regions that disappear as soon as you release the mouse button. The graphical window has scrollbars that you can manipulate, although there are no objects inside the scroller to scroll at this point.

We began by creating a scroller with `this` (the window) as the enclosure:

```
itsScroller = new CScroller(this, 1000, 1000);
```

The dimensions of the scroller are equal to those of the window's frame as returned by `GetFrame`. The virtual scrolling area has a size of 1000 by 1000 pixels. The second line initializes several scroller attributes by calling `IScroller`.

See Also: For information about its parameters, see the `CScroller` section in the *XVT-Power++ Reference*.

XVT-Power++ provides a class called `CGlue` that allows us to glue all four sides of a view to its enclosure to achieve the effect of stretching and shrinking of the view as the enclosure is resized. We do not have to create a `CGlue` object; XVT-Power++ does this internally. To give the scroller glue capabilities so that it sizes with the window, we added the line shown here in boldface type to the `CDrawWin` source file.

```
... code ...
itsScroller->IScroller(TRUE, TRUE, TRUE, 10, 50);
itsScroller->SetGlue(ALLSTICKY);
```

Try sizing the Draw window now. Notice that the scroller sizes as well. You may want to comment out the `SetGlue` line and recompile the program to test the behavior of the scroller when it does not have glue capabilities and the window is resized.

Once scrolling was enabled, we created a sketchpad using the `CSketchPad` class, which allows the user to drag out areas on the screen with the mouse. To construct the sketchpad, we entered the following line immediately after the lines initializing the scroller:

```
itsSketchPad = new CSketchPad(itsScroller, itsScroller ->
                    GetVirtualFrame());
```

The sketchpad's enclosure is `itsScroller`. The sketchpad's dimensions are equal to those of the scroller's virtual scrolling frame returned by `GetVirtualFrame`—in this case, 1000 by 1000.

We set the sketchpad's command with this line:

```
itsSketchPad->SetCommand(SKETCHcmd);
```

Recall that objects generate commands upon being clicked (see the Hello World example). The sketchpad generates the `SKETCHcmd` when the user sketches an object upon it.

3.4.6. Creating Graphical Objects Inside the Drawing Window

While we can use the mouse to sketch out regions inside the drawing window, we cannot yet create any objects. Our goal in this section is to enable a user to select a type of object from the Objects menu and then create the object when sketching a region in the scroller. The objects created will be scrollable, sizable, and draggable.

We begin by adding some new methods and data members to the `CDrawWin` class.

```
class CDrawWin : public CWindow
{
    public:
        ... code ...
        virtual void DoMenuCommand(MENU_TAG
                theMenuItem,BOOLEAN isShiftKey,
                BOOLEAN isControlKey);

    private:
         ... code ...

    // keep track of object type selection
        unsigned itsMenuObject;
    // build a new sketched object
        CView* BuildObject(void);
};
```

We add some extra initializing code to the CDrawWin constructor:

```
CDrawWin::CDrawWin(CDocument *theDocument ...
{
    itsMenuObject = RECTANGLEid;

    // Setup the appropriate menus:
    win_menu_enable(GetXVTWindow(), M_FILE_NEW, TRUE);
    win_menu_enable(GetXVTWindow(), OBJECTMenu, TRUE);

    ... code ...
}
```

Then we define the DoMenuCommand to handle menu events:

```
void CDrawWin::DoMenuCommand(MENU_TAG theMenuItem,
                BOOLEAN isShiftKey, BOOLEAN isControlKey)
{
    switch (theMenuItem)
    {
        case RECTANGLEid:
        case OVALid:
            win_menu_check(GetXVTWindow(), itsMenuObject, FALSE);
            win_menu_check(GetXVTWindow(), theMenuItem, TRUE);
            itsMenuObject = theMenuItem;
            break;

        default:
            CWindow::DoMenuCommand(theMenuItem, isShiftKey, isControlKey);
            break;
    }
}
```

We define the DoCommand to trap user sketch actions:

```
void CDrawWin::DoCommand(long theCommand, void* theData)
{
    switch (theCommand)
    {
        case SKETCHcmd:
            BuildObject();
            break;

        default:
            CWindow::DoCommand(theCommand,theData);
    }
}
```

Finally, we define BuildObject to create the graphical views:

```
CView* CDrawWin::BuildObject(void)
{
    CView* newObject;

    // First create the object:
    switch (itsMenuObject)
    {
        case RECTANGLEid:
            newObject = new CRectangle(itsScroller,
                            itsSketchPad->GetSketchedRegion());
            break;

        case OVALid:
            newObject = new COval(itsScroller,
                            itsSketchPad->GetSketchedRegion());
            break;
    }

    // Now set special attributes:
    newObject->SetSizing(TRUE);
    newObject->SetDragging(TRUE);
    newObject->SetEnvironment(CEnvironment(COLOR_BLACK,
        COLOR_RED, COLOR_RED));
    newObject->SetCommand(itsMenuObject);
    newObject->DoDraw();

    return newObject;
}
```

After we recompile and execute the program, we can drag the mouse inside the graphical window to draw ovals and rectangles, as shown in Figure 7. Let's consider the specifics of how we achieved our goals in this phase of the Draw program's development.

First, we added a private data member called `itsMenuObject` that serves as an indicator of the current selection from the Objects menu. When we defined the application menubar, we assigned the "Rectangle" menu item an initial checked state:

```
MENU OBJECTMenu "Objects"
    ITEM RECTANGLEid "Rectangle" CHECKED
    ITEM OVALid "Oval" CHECKABLE
```

That is why we initialize `itsMenuObject` accordingly inside the window's constructor. While we are at it, we also enable the Objects menu and the "New" item on the File menu, as follows:

```
itsMenuObject = RECTANGLEid;

// Setup the appropriate menus:
win_menu_enable(GetXVTWindow(), M_FILE_NEW, TRUE);
win_menu_enable(GetXVTWindow(), OBJECTMenu, TRUE);
```

In order to handle the user selections from the Object menu, we defined the `DoMenuCommand` method. This is a virtual method contained by all classes in the XVT-Power++ application framework. `DoMenuCommand` is called when the user selects an item from the menubar. In our implementation, we trap the selection of items in the Objects menu as follows:

```
case RECTANGLEid:
case OVALid:
    win_menu_check(GetXVTWindow(),itsMenuObject,FALSE);
    win_menu_check(GetXVTWindow(), theMenuItem,TRUE);
    itsMenuObject = theMenuItem;
    break;
```

We respond by unchecking the previous menu selection and checking the new one. In the default case, the inherited `DoMenuCommand` is called, as is true of the `DoCommand` method. Calling the inherited method ensures that all other menu commands are propagated up the XVT-Power++ object hierarchy.

We also added a case to the `DoCommand` method to handle the messages from the sketchpad:

```
case SKETCHcmd:
    BuildObject();
    break;
```

Remember that we set the sketchpad's command to SKETCHcmd so that this command is generated when the user sketches a new region. We respond by calling the BuildObject method, which is in charge of creating the actual graphical objects. Let's analyze what this method does:

1. First, we switch on the value of the internal itsMenuObject variable, and create the appropriate object.
2. The COval and CRectangle objects have similar constructions.
   - The first parameter, itsScroller, indicates the enclosure of the new object.
   - The next parameter, itsSketchPad->GetSketchedRegion, indicates the size of the new object and is equivalent to the size and location of the region sketched by the user.
3. Finally, we set attributes for the newly created object. Notice that the same methods can be called on the pointer newObject regardless of whether it pointing to a CRectangle or a COval. Let's examine each of the attributes given to the objects. The first two calls are:

```
newObject->SetSizing(TRUE);
newObject->SetDragging(TRUE);
```

Setting sizing and dragging to TRUE enables the user to size and drag the objects using the mouse. Next is this call:

```
newObject->SetEnvironment(CEnvironment(COLOR_BLACK,
                                       COLOR_RED, COLOR_RED));
```

SetEnvironment allows us to set such attributes of a view as colors, fonts, brush patterns, and pen widths. Any CView can have its own CEnvironment. If a view's environment is not set explicitly, the view inherits the environment settings of its enclosure. The CEnvironment class and its use inside XVT-Power++ is discussed in the *XVT-Power++ Reference*.

Finally, the following call sets the command of each object (the reason for this will become apparent later):

```
newObject->SetCommand(itsMenuObject);
```

BuildObject illustrates one of the beauties of the polymorphism provided by XVT-Power++ CView classes: knowledge of the interface to one class can usually be transferred as knowledge of the interface to most other classes.

3.4.7. Building a Textual View of a Document's Data

We are ready to build the second part of the Draw program, which will allow a document to create a second view of its data. The *data* in this case are the shapes that have been dynamically created. We have a graphical view of that data and will implement a textual view as well. This feature may not be crucial to a drawing application, but it illustrates the concept of multiple views for a single document.

Our specific goal is to propagate the changes in the document's data to the textual view. These changes can be in the form of a new object created by a user or of existing objects being sized and moved.

We will design the Draw program to generate `DoCommand` messages whenever a user changes the graphical view. These commands are trapped by the `CDrawDoc` object, which will take care of updating the textual view.

First, we add the code inside of `CDrawWin` to generate events when the graphical view changes. We modify the `DoCommand` as follows:

```
void CDrawWin::DoCommand(long theCommand,void* theData)
{
    switch (theCommand)
    {
        case SKETCHcmd:
            DoCommand(NEWObjectCmd,BuildObject());
            break;

        default:
            CWindow::DoCommand(theCommand,theData);
    }
}
```

Sketch commands are now handled by generating a new command. The command ID is indicated by `NEWObjectCmd`. Its value is defined inside **Draw.h**:

```
#define NEWObjectCmd  109 /* Generated when a new view is created. */
```

Now that the command can be generated, we add the code (in the `CDrawDoc` source file) to trap it at the document level:

```
void CDrawDoc::DoCommand(long theCommand,void* theData)
{
    switch (theCommand)
    {
        case NEWObjectCmd:
            SetSave(TRUE);
            DisplayText((CView*) theData);
            break;

        case WFSizeCmd:
            SetSave(TRUE);
            DisplayText((CView*) theData);
            break;
```

```
default:
            CDocument::DoCommand(theCommand,theData);
            break;
    }
}
```

We add the new method called `DisplayText`, as well as the `CView` class to the `CDrawDoc` header file.

```
class CWindow;
class CDrawWin;
class NScrollText;
class CView;

class CDrawDoc : public CDocument
{
    public:

        ... code ...
        void DisplayText(CView* theNewObject);
        ... code ...
};
```

We define the `DisplayText` method as follows:

```
void CDrawDoc::DisplayText(CView* theNewObject)
{
    PwrAssert((itsText!=NULL) && (theNewObject!=NULL),
            1, "Invalid call to DisplayText()");

    itsText->Suspend();

    CString object(int(theNewObject->GetCommand()));
    itsText->AddParagraph("Object Type: " + object, USHRT_MAX);

    char top[20], left[20], right[20], bottom[20];
    CRect rect = theNewObject->GetFrame();

    left[0] = NULL;
    sprintf(left,"%d", int(rect.Left()));
    top[0] = NULL;
    sprintf(top,"%d", int(rect.Top()));
    right[0] = NULL;
    sprintf(right,"%d", int(rect.Right()));
    bottom[0] = NULL;
    sprintf(bottom,"%d", int(rect.Bottom()));

    CString frame = "Object Frame: ";
    itsText->AddParagraph(frame + left + " " + top + " " +
                    right + " " + bottom, USHRT_MAX);
    itsText->AddParagraph("", USHRT_MAX);

    itsText->Resume();
}
```

We recompile and execute the new program. Opening a new document invokes two windows, and the creation of new objects is tracked dynamically in the text window. When the views are sized or moved, their new coordinates are displayed in the text window as well. The "Save" and "Save As" File menu items become enabled/

disabled, depending on whether the document's data has changed and needs saving.

Let's take a closer look at how we achieved this final goal in the Draw program. One way that the graphical view can change is through the creation of a new shape. Thus, we generate a command each time the user drags out a new object by slightly modifying the DoCommand handling of SKETCHcmd inside CDrawWin:

```
DoCommand(NEWObjectCmd,BuildObject());
```

The return value of BuildObject is sent along with the NEWObjectCmd command. Recall that BuildObject returns a pointer of type CView* to the newly created object. Along with DoCommand messages, we can pass any type of data in the form of a void*. The object passed in the DoCommand will be used by the document to obtain information needed for the textual view.

XVT-Power++ directs DoCommand messages so that they travel from an object to its parent. Here, the command is generated by CDrawWin, and it is trapped by its parent, CDrawDoc, in the following code:

```
case NEWObjectCmd:
    SetSave(TRUE);
    DisplayText((CView*) theData);
    break;
```

The NEWObjectCmd case does two things. First, it calls SetSave(TRUE), meaning that the data in the document has changed and that the document needs saving. Next, it calls a new CDrawDoc method called DisplayText and passes along the pointer to the newly created object. Notice that we had to cast theData back to a CView*. For commands that pass along objects, it is important that the type of the data be well-defined. Otherwise, the cast used would be unsafe.

Just as XVT-Power++ generates commands when views are clicked, it also generates commands when views are selected, dragged, sized, and deselected. The WFSizeCmd command is generated when a user drags or sizes a view. With the WFSizeCmd, we also get a pointer to the object that was moved or sized. We trapped the WFSizeCmd command inside the DoCommand and executed steps that are identical to those in the NEWObjectCmd.

In both of the DoCommand cases, we call a new method called DisplayText to actually display the textual information. The first line of the DisplayText method asserts that itsText has been created and that the object passed is not NULL. The rest of the method adds text to the scroll text created inside the text window. Notice that

updates to `itsText` are suspended and then resumed before the function is exited. This speeds up the insertion of text.

The following two lines insert the type of the object into the window:

```
CString object(int(theNewObject->GetCommand()));
itsText->AddParagraph("Object Type: " + object, USHRT_MAX);
```

The first line creates a string using a string resource defined using `theNewObject`'s command value. Remember that we wrote the following line inside `CDrawWin::BuildObject`:

```
newObject->SetCommand(itsMenuObject);
```

This line gives the object one of two commands: `RECTANGLEid` or `OVALid`. We used these two commands to define two string resources inside **Draw.url**:

```
/* Define string resources used for object type names: */

STRING OVALid "Oval"
STRING RECTANGLEid "Rectangle"
```

Thus, the line
"`CString object(int(theNewObject->GetCommand()));`"
creates a new string using a string constructor that looks up the string in the resource table.

See Also: For details, see the section on `CString` in the *XVT-Power++ Reference*

What we have actually accomplished here is to obtain the object's type name through a dynamic type identification scheme. The code first converts the numerical data provided by `theNewObject->GetFrame` into strings that are concatenated and added to the text.

Note: In the future, this kind of type identification may be built into every class. There is even a heated discussion in the C++ community of making this a standard feature. For now we settled for using a command value and string resource scheme.

The rest of the code adds information about the view's coordinates:

```
char top[20], left[20], right[20], bottom[20];
CRect rect = theNewObject->GetFrame();
```

```
left[0] = NULL;
sprintf(left,"%d",rect.Left());
top[0] = NULL;
sprintf(top,"%d",rect.Top());
right[0] = NULL;
sprintf(right,"%d",rect.Right());
bottom[0] = NULL;
sprintf(bottom,"%d",rect.Bottom());

CString frame = "Object Frame: ";
itsText->AddParagraph(frame + left + " " + top + " " +
    right + " " + bottom, USHRT_MAX);
```

3.4.8. Summary

While this concludes our discussion of the Draw program, there are clearly more features that would be useful: the capacity to open saved text files and recreate the objects saved, as well as the capacity to modify the text view and have the graphical view change accordingly. These features are a bit more complicated because they involve parsing text and translating it into creation or manipulation of XVT-Power++ objects. We are adding these features to HardHat so that actual code or resource definitions are generated and used to build XVT-Power++ windows and other views. For now, we leave this final suggestion as an exercise for you if you want further challenges and would like to extend what you have learned here about XVT-Power++.

Following is a listing of the entire Draw program.

CDrawApp.h

```
#ifndef CDrawApp_H
#define CDrawApp_H

#include "PwrDef.h"
#include CApplication_i

class CDrawApp : public CApplication
{
    public:
        CDrawApp(void);

        virtual void StartUp(void);
        virtual void DoCommand(long theCommand,void*
theData=NULL);

        virtual BOOLEAN DoNew(void);
        virtual BOOLEAN DoOpen(void);

        virtual void SetUpMenus(void);
};

#endif CDrawApp_H
```

CDrawApp.cxx

```
#include "CDrawApp.h"
#include "CDrawDoc.h"

////////////////////////////////////////////////////////////////////////////
CDrawApp::CDrawApp(void) : CApplication()

{

}
////////////////////////////////////////////////////////////////////////////
void CDrawApp::StartUp(void)
{
    CApplication::StartUp();

    about_box();
}
////////////////////////////////////////////////////////////////////////////
void CDrawApp::DoCommand(long theCommand,void* theData)
{
    switch (theCommand)
    {
        default: CApplication::DoCommand(theCommand,theData);
        break;
    }
}
////////////////////////////////////////////////////////////////////////////
BOOLEAN CDrawApp::DoNew(void)
{
    if (DoClose())
    {
        CDrawDoc* aDoc = new CDrawDoc(this, GetNumDocuments()+1);
        PwrAssert(aDoc, 311, "Creation of CDrawDoc failed");
        return aDoc->DoNew();
    }
}
////////////////////////////////////////////////////////////////////////////
BOOLEAN CDrawApp::DoOpen(void)
{
    CDrawDoc* aDoc = new CDrawDoc(this, GetNumDocuments()+1);
    PwrAssert(aDoc, 311, "Creation of CDrawDoc failed");
    return aDoc->DoOpen();
}
////////////////////////////////////////////////////////////////////////////
void CDrawApp::SetUpMenus(void)
{
    win_menu_enable(TASK_WIN, M_FILE_NEW, TRUE);
}
```

CDrawDoc.h

```
#ifndef CDrawDoc_HXX
#define CDrawDoc_HXX


#include "PwrDef.h"
#include CDocument_i

class CWindow;
class CDrawWin;
class NScrollText;
class CView;
```

```
class CDrawDoc : public CDocument
{
    public:

        CDrawDoc(CApplication *theApplication,int theId );
        virtual void BuildWindow(void);
        virtual void DoCommand(long theCommand,void* theData=NULL);

        virtual void UpdateMenus(void);
        virtual BOOLEAN DoSave(void);

    private:

        void DisplayText(CView* theNewObject);

        CDrawWin* itsDrawWin;
        CWindow* itsTextWin;
        NScrollText* itsText;
};

#endif CDrawDoc_HXX
```

CDrawDoc.cxx

```
#include "PwrDef.h"
#include NScrollText_i
#include CWindow_i

#include "CDrawDoc.h"
#include "CDrawWin.h"
#include "Draw.h"

#ifdef STDH
#include <stream.h>
#else
#include <fstream.h>
#endif

////////////////////////////////////////////////////////////////////////
CDrawDoc::CDrawDoc(CApplication *theApplication,int theId ) :
                CDocument(theApplication,theId),
                itsDrawWin(NULL), itsTextWin(NULL), itsText(NULL)
{

}
////////////////////////////////////////////////////////////////////////

void CDrawDoc::DoCommand(long theCommand,void* theData)
{
    switch (theCommand)
    {
        case NEWObjectCmd:
            SetSave(TRUE);
            DisplayText((CView*) theData);
            break;

        case WFSizeCmd:
            SetSave(TRUE);
            DisplayText((CView*) theData);
            break;
```

```
        default:
            CDocument::DoCommand(theCommand,theData);
            break;
    }
}
////////////////////////////////////////////////////////////////////////////
void CDrawDoc::BuildWindow(void)
{
    PwrAssert((itsDrawWin==NULL) && (itsTextWin==NULL),1,
            "BuildWindow() called with existing windows");

    // First build a graphical view:

    itsDrawWin = new CDrawWin(this, CRect(20,20,300,300), "Draw Window",
                WSF_SIZE|WSF_ICONIZABLE|WSF_HSCROLL|
                WSF_VSCROLL,      W_DOC, MENUBarRid);

    // Next build a textual view:

    itsTextWin = new CWindow(this, CRect(120,20,420,280),
                "Log Window",
                WSF_SIZE|WSF_ICONIZABLE|WSF_HSCROLL
                WSF_VSCROLL);

    itsText = new NScrollText(itsTextWin, TX_BORDER|TX_READONLY|
                    TX_AUTOVSCROLL);
}
////////////////////////////////////////////////////////////////////////////

void CDrawDoc::UpdateMenus(void)
{
    CWindow* win;
    CIterator doTo(itsWindows);
    while (win = (CWindow*) doTo.Next())
    {
        win_menu_enable(win->GetXVTWindow(), M_FILE_SAVE_AS,
                NeedsSaving());

    if (itsXVTFilePointer)
        win_menu_enable(win->GetXVTWindow(), M_FILE_SAVE, NeedsSaving());
    }
}

////////////////////////////////////////////////////////////////////////////
BOOLEAN CDrawDoc::DoSave(void)
{
     // Handle File menu Save

    if (!CDocument::DoSave())
        return FALSE;

    // Save the file:
    if (itsText)
    {
        ofstream fileStream(itsXVTFilePointer->name);
        fileStream << itsText->GetText();
        fileStream.close();
    }
    itsDrawWin->SetTitle(itsXVTFilePointer->name);
    if (itsTextWin)
        itsTextWin->SetTitle(itsXVTFilePointer->name);
return TRUE;
}
```

```
//////////////////////////////////////////////////////////////////////////
void CDrawDoc::DisplayText(CView* theNewObject)
{
     PwrAssert((itsText!=NULL) && (theNewObject!=NULL),
               2, "Invalid call to DisplayText()");

     itsText->Suspend();

     CString object(int(theNewObject->GetCommand()));
     itsText->AddParagraph("Object Type: " + object, USHRT_MAX);

     char top[20], left[20], right[20], bottom[20];
     CRect rect = theNewObject->GetFrame();

     left[0] = NULL;
     sprintf(left,"%d", int(rect.Left()));
     top[0] = NULL;
     sprintf(top,"%d", int(rect.Top()));
     right[0] = NULL;
     sprintf(right,"%d", int(rect.Right()));
     bottom[0] = NULL;
     sprintf(bottom,"%d", int(rect.Bottom()));

     CString frame = "Object Frame: ";
     itsText->AddParagraph(frame + left + " " + top + " " +
                    right + " " + bottom, USHRT_MAX);
     itsText->AddParagraph("", USHRT_MAX);

     itsText->Resume();
}
```

CDrawWin.h

```
#ifndef CDrawWin_HXX
#define CDrawWin_HXX

#include "CWindow.h"

class CScroller;
class CSketchPad;

class CDrawWin : public CWindow
{
     public:

     CDrawWin(CDocument *theDocument,
              const CRect& theRegion,
              const CString& = NULLString,
              long theWindowAttributes = WSF_NONE,
              WIN_TYPE theWindowType = W_DOC,
              int theMenuBarId = MENU_BAR_RID);

     virtual void DoCommand(long theCommand,void* theData=NULL);
     virtual void DoMenuCommand(MENU_TAG theMenuItem,
                    BOOLEAN isShiftKey, BOOLEAN isControlKey);

     private:

          CScroller* itsScroller;
          CSketchPad* itsSketchPad;
          CList itsObjectList;
          unsigned itsMenuObject;
          CView* BuildObject(void);
};
```

```
#endif CDrawWin_HXX
```

CDrawWin.cxx

```
#include "PwrDef.h"
#include CScroller_i

#include CSketchPad_i
#include CRectangle_i
#include COval_i
#include CDocument_i

#include "CDrawWin.h"
#include "Draw.h"

////////////////////////////////////////////////////////////////////////
CDrawWin::CDrawWin(CDocument *theDocument,
                   const CRect& theRegion,
                   const CString& theTitle,
                   long theAttributes,
                   WIN_TYPE theWindowType,
                   int theMenuBar)

: CWindow(theDocument, theRegion, theTitle,
             theAttributes, theWindowType, theMenuBar)
{
     // Add any code to create sub-objects here.

     itsMenuObject = RECTANGLEid;

     // Setup the appropriate menus:
     win_menu_enable(GetXVTWindow(), M_FILE_NEW, TRUE);
     win_menu_enable(GetXVTWindow(), OBJECTMenu, TRUE);

     // Create the scroller:
     itsScroller = new CScroller(this, 1000,1000);
     itsScroller->IScroller(TRUE, TRUE, TRUE, 10, 50);
     itsScroller->SetGlue(ALLSTICKY);

     // Create a sketchpad inside the scroller:
     itsSketchPad = new CSketchPad(itsScroller,
                         itsScroller->GetVirtualFrame());
     itsSketchPad->SetCommand(SKETCHcmd);
     itsSketchPad->SetSketchEverywhere(FALSE);
}
////////////////////////////////////////////////////////////////////////
void CDrawWin::DoCommand(long theCommand,void* theData)
{
     switch (theCommand)
     {
          case SKETCHcmd:
          DoCommand(NEWObjectCmd,BuildObject());
          break;

     default:
     CWindow::DoCommand(theCommand,theData);
     }
}
```

```
//////////////////////////////////////////////////////////////////////////
void CDrawWin::DoMenuCommand(MENU_TAG theMenuItem,
                  BOOLEAN isShiftKey, BOOLEAN isControlKey)
{
     switch (theMenuItem)
     {
          case RECTANGLEid:
          case OVALid:
          win_menu_check(GetXVTWindow(), itsMenuObject, FALSE);
          win_menu_check(GetXVTWindow(), theMenuItem, TRUE);
          itsMenuObject = theMenuItem;
          break;

     default:
          CWindow::DoMenuCommand(theMenuItem, isShiftKey, isControlKey);
          break;
     }
}
//////////////////////////////////////////////////////////////////////////
CView* CDrawWin::BuildObject(void)
{
     CView* newObject;

     // First create the object:

     switch (itsMenuObject)
     {
          case RECTANGLEid:
               newObject = new CRectangle(itsScroller,
                         itsSketchPad->GetSketchedRegion());
          break;

     case OVALid:
          newObject = new COval(itsScroller,
                    itsSketchPad->GetSketchedRegion());
          break;
     }

     // Now set special attributes:

     newObject->SetSizing(TRUE);
     newObject->SetDragging(TRUE);
     newObject->SetEnvironment(CEnvironment(COLOR_BLACK, COLOR_RED,
                    COLOR_RED));
     newObject->SetCommand(itsMenuObject);
     newObject->DoDraw();

     return newObject;
}
```

CDrawStartup.cxx

```
#include "CDrawApp.h"
#include "Draw.h"

extern "C"
{
     void main(int argc, char* argv[])
     {
          CDrawApp theApplication;
          theApplication.Go(argc, argv, MENUBarRid, ABOUTBOXRid, "Draw",
                    "Draw",   "Draw");
     }
}
```

Draw.url

```
#define APPNAME draw
#define QAPPNAME "Draw"

#include "url.h"
#include "PwrURL.h"

#include "Draw.h"

/* Definition of the about box dialog box: */
DIALOG ABOUTBOXRid 100,100,390,120
     BUTTON DLG_CANCEL, 330, 75, 35, 25 "OK" DEFAULT
     TEXT ABOUTText1 15, 15, 360, 15 "** Draw: A Power++ Example Application **"
     TEXT ABOUTText2 15, 40, 360, 15 "Program illustrates Power++ programming"
     TEXT ABOUTText3 15, 60, 360, 15 "and is part of Power++ tutorial set."

/* Define the application menubar: */

MENUBAR MENUBarRid

MENU MENUBarRid
     DEFAULT_FILE_MENU
     SUBMENU OBJECTMenu "Objects" DISABLED

MENU OBJECTMenu "Objects"
     ITEM RECTANGLEid "Rectangle" CHECKED
     ITEM OVALid "Oval" CHECKABLE

/* Define string resources used for object type names: */

STRING OVALid "Oval"
STRING RECTANGLEid "Rectangle"
```

3.5. Chapter Summary

Having completed all three exercises in this tutorial, you now have a working knowledge of how to build an XVT-Power++ program.

In developing the "Hello World" program, you learned how to compile XVT-Power++'s shell application and bring up a window. When you added textual views to the window, you encountered the `NLineText` and `CText` classes and explored how objects of the `CView` class can interact with the user and respond to events. You learned about one of the channels of communication that allows different XVT-Power++ objects to send messages to each other: the `DoCommand` method chain

The Edit program builds upon the "Hello World" program, again making use of XVT-Power++'s shell classes. In the first exercise, you learned how to bring up a window and add views to it. In the Edit exercise, you explored how to create multiple documents, each of which can have multiple views. You developed a text file editor that can bring up a window, open a file and display it inside a window for editing, edit the file, save the file back to disk, and save and close before exiting the program.

Finally, the Draw exercise familiarized you with the structure of an *entire* XVT-Power++ application while you explored such specific features of XVT-Power++ as resource creation, view nesting, scrolling, view manipulation, data propagation, and menubar command handling. You learned how to create multiple documents and to build multiple views of a document's data. You also learned how to set up the application and document menubars.

You are now ready to experiment with some small programs. We recommend that you choose a few classes from the *XVT-Power++ Reference*, instantiate them, and use them. Before you start a serious XVT-Power++ application, however, read the other chapters of this *Guide* to get an in-depth knowledge of XVT-Power++'s functionality.

4

APPLICATIONS

4.1. Introduction

`CApplication`, an abstract XVT-Power++class that you must override for each application that you write, creates and manages the application object. The application object resides at the top level of the XVT-Power++ application framework, performing several application management functions. It controls the program from starting to quitting: its direction, its modes, different documents that are open at different points, and so on. The application object initiates any object the program needs when it starts running and cleans up after the program upon shutdown. It also sets up the global objects and global data that are provided to all objects through `CBoss`. Moreover, the application class is responsible for creating and managing the application's documents.

This chapter describes the start-up and shutdown mechanisms in `CApplication` and surveys the tasks performed at the application level.

4.2. Application Start-up and Shutdown

The application object for each program is created in the `main` function, which is located in the **CStartUp** source file. The `main` function creates an application object, giving it the information that it needs upon creation, and then invokes a `Go` method. An example of this procedure is shown here:

```
void main(int argc, char* argv[])
{
    CMyApptheApplication;
    theApplication.Go(argc, argv, MY_MENU_BAR_RID,
            ABOUTBOX, "BaseName", "Application Name",
            "Task Title");
}
```

The definition of Go is as follows:

```
void CApplication::Go(int argc, char *argv[],
                short theMenuBarId,
                short theAboutBoxId,
                char *theBaseName,
                char *theApplicationName,
                char *theTaskTitle)
```

`arcg` and `argv` are passed literally as they are supplied to `main`. `theMenuBarId` is the resource ID number of the application's menubar, `theAboutBoxId` is the resource ID number of the About box, and `theBaseName` is the name of the application stored on disk. The string passed to `theApplicationName` is used as the title of the windows belonging to the application. Finally, the string passed to `theTaskTitle` is used as the title of the application's task window, if there is a task window.

Once the `Go` method is called, the XVT-Power++/XVT Portability Toolkit system takes over. Execution is never returned to the `main` function. When the application starts executing, the XVT-Power++ library hooks up to the main event loop of the XVT Portability Toolkit system. When both XVT-Power++ and the XVT Portability Toolkit have finished initializing, the application's `StartUp` method is called. As its name indicates, `StartUp` handles start-up activities for the application. It is a virtual method that you can override, but keep in mind that it must be called by any derived classes if it is overridden. That is, `CApplication::StartUp` is called before anything else inside of `StartUp`. After that, you specify what you want your application to do upon start-up: create certain windows, open specified documents, connect to a database, and so on.

As the application runs, the user opens documents by selecting menu items. At some point, the user will perform an action indicating that he or she is ready to quit the program, perhaps by selecting "Quit" from the File menu. The `CApplication::ShutDown` method is called to perform application-dependant shutdown tasks, such as closing any files and disconnecting from a database. Normally, `ShutDown` does not have to perform any actions related to XVT-Power++ objects. For example, if windows are up, they are closed and deleted automatically through XVT-Power++. If you override `ShutDown`,

keep in mind that the first thing it should do is call the inherited ShutDown method.

4.3. Tasks Handled at the Application Level

When an XVT-Power++ application starts, it initializes various program defaults, such as enabling or disabling some menu items on the menubar and bringing up a default window such as a splash screen or a dialog box.

One of the responsibilities of the CApplication class is to manage global objects and global data for the application. Objects managed by the application include:

- the global environment that is used by any view object, unless it has a specific environment
- the global objects inside the global class library, CGlobalClassLib, that contain information about the state of the application
- the desktop, which manages window layout on the screen
- some event handler objects that channel all events coming to the application to the appropriate objects, such as windows or views. Some of these event handler objects can be accessed through the application, but most of them can be accessed through the CGlobalClassLib object. Every object that inherits from CBoss(i.e., any object in the application framework), has access to those global objects through theCRadioGroup pointers

CApplication contains a number of "Do-" methods that are called automatically when different menu actions are selected. For example, when a user selects "Open", "New", "Save", and so on, from the menubar, the application object automatically takes care of these operations, channeling them through the DoOpen, DoSave, DoNew methods and so on. You can override these "Do-" methods to perform such actions as you choose. DoOpen typically creates a document and sends it a DoOpen message. DoNew creates a new document and sends it a DoNew message. DoSave sends a DoSave message to all of the application's documents.

DoClose, another mechanism automatically provided by the application, closes all open documents. Similarly, if at any point the user elects to cancel a certain operation, DoClose seizes this operation. In addition, the application includes a number of document management functions. It has functions for finding open

documents, closing all documents, and adding and removing documents from the application.

See Also: For details on `CApplication`, refer to the `CApplication` section in the *XVT-Power++Reference*. For an example of the minimum methods you must override in order to derive your own application class, see `CShellApp`.

4.4. Chapter Summary

In this chapter, we have examined `CApplication`'s procedures for starting a program and shutting it down, noted the global objects that are managed by `CApplication`, and surveyed the automatic mechanisms provided at the application level. In Chapter 5, we explore in detail how XVT-Power++ accesses and manages data at the document level.

5

DOCUMENTS

5.1. Introduction

Each application normally needs to manage data in some form, whether it be stored as a file or as a record in a database. The data can be accessed in different ways, perhaps through the network from a server process. The XVT-Power++ class that is in charge of accessing and managing data is `CDocument`, an abstract class from which you must derive and instantiate your own specific document classes. The document is the link between the application level and the view level of the XVT-Power++ application framework. The `CApplication` object instantiates and manages `CDocument` objects, which in turn instantiate and manage `CView` objects for displaying data (see Figure 9).

This chapter begins by explaining how data is propagated within the XVT-Power++ system. Then it describes in detail the different tasks performed at the document level in XVT-Power++'s application framework: accessing data, building windows, managing data, and managing windows.

5.2. Data Propagation

`CDocument` can create totally different type of views for viewing the same set of data in different display formats. When a document has constructed multiple types of views to display the same set of data, it is in charge of coordinating changes in the data and updating the views to reflect the changes. Such coordination is called *data propagation.*

XVT-Power++ propagates data through its `DoCommand` mechanism, known more descriptively as the `DoCommand chain`. Typically, if the data inside a view changes, the view generates a `DoCommand` to the

document. In response, the document sets its "needs saving" state to TRUE and may update the data display in several of its windows to reflect the change. Similarly, if a window associated with a particular document needs to send messages to other windows of that document, it sends a DoCommand to the document and lets the document take care of propagating that message to all of the other windows. Thus, in a sense, the document acts as a server to all of its client windows. The windows can request it to do something or pass on messages to other windows by simply generating a DoCommand. A DoCommand naturally flows upward through the event structure of XVT-Power++, starting from a view, perhaps moving through a series of enclosures to the window containing that view, and finally arriving at the document that owns the window.

Let's look at how a document displays data in multiple windows. Suppose there is an application containing a graph and a spreadsheet that share the same data but display it in very different formats. The graph merely displays the information. The spreadsheet not only displays it but also provides the means to edit it directly. In XVT-Power++, the spreadsheet and the graph will have both been created by a CDocument object that they share. This document object manages the data and stores the information for both windows.Thus the document manages the data that is used by the windows.

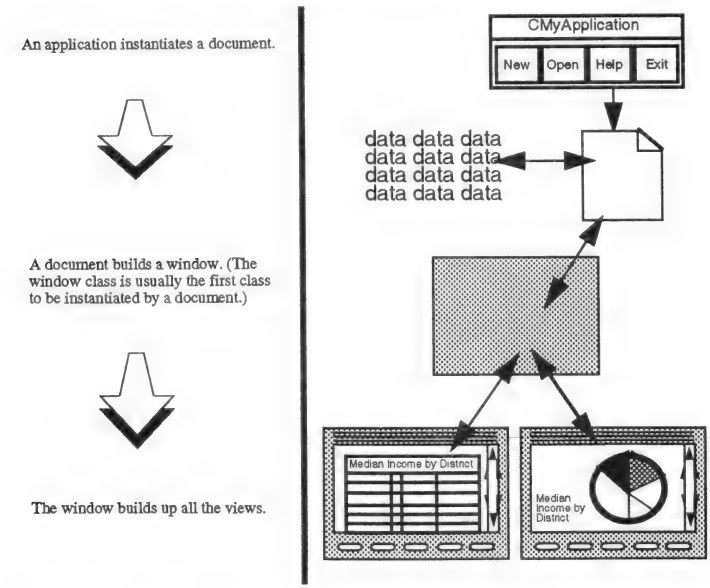


Figure 9. Order in Which Objects are Created Within the Application

Another use of a document is communication. When you make a change inside the spreadsheet window and then click on a button to commit the changes, you also want to notify the graph window to refresh its graph. In this case there are two windows that need to communicate with each other. The need for a central means of communication becomes even more critical when several windows sharing the same data need to communicate with each other.

There are different ways to handle the communication. One way would be for the spreadsheet window to communicate directly with the graph window, informing it of the need to refresh its graph. Another way would be through their common `CDocument` object. In this case, the spreadsheet informs the document object of the change, which it has to do anyway since the commit button has been pressed and the document object is managing the data. The document then communicates the change to the graph and to any other window that needs to know about the change because it is sharing the data. In response, the windows update their graphical representations to reflect the new data.

Using the `CDocument` object to communicate changes is very effective when more than two windows are sharing the same data. They are sharing not only the place where the data is accessed (the document) but also an object that is orchestrating the communication of data changes—all in one place.

5.3. Tasks Handled at the Document Level

The following sections describe in detail the data access and data management features provided in XVT-Power++'s `CDocument` class.

5.3.1. Accessing Data

XVT-Power++ allows different mechanisms for creating new documents. Typically, a document object is instantiated when the user selects "New" or "Open" from the File menu. Some applications may create several documents and open them by default when the application comes up. In XVT-Power++, to *open* a document means to instantiate a `CDocument` object, and to *close* a document means to *delete* or *destroy* the `CDocument` object.

When a document is opened, its first task is to get access to data. The way a document does this partly depends on whether it is opened with or without data. A new document, of course, has no data associated with it at first. By default, to create a new document, you

call the CDocument DoNew method. Part of DoNew's task might be to create a new document that contains no data but is simply on the screen, ready to display whatever data the user specifies. For example, the user might create a new spreadsheet that does not yet have any data to manage. In this case, the CDocument object would instantiate a window with an empty spreadsheet. On the other hand, if the CDocument object instantiates a window that displays some initial data, call the CDocument DoOpen instead of DoNew. You can override DoOpen so that it retrieves data from any specified place and sets up the views to display the data.

It is important that you define DoNew and DoOpen because in many applications documents are created through the File menu "Open" or "New" menu items. It is also important that you define the appropriate actions to take inside those methods.

When you define the document interface for your application by deriving new document classes from CDocument, you can add parameters to the constructor that tell the document what data it should access, perhaps a pointer to a filename or an indication of which database record to get. Next, we recommend that you place the actual code for initializing the data in the CDocument DoNew and DoOpen methods. If you call the inherited methods as well to take advantage of their default behavior, another method, named BuildWindow, is also called.

5.3.2. Building Windows

Once you have accessed some data, probably through the CDocument constructor, and have opened it—that is, have created an empty document—you are ready to build a window to display the data. Every document must define how to build a window. The CDocument BuildWindow method handles this task. This method is declared as a pure virtual at the CDocument level. BuildWindow notifies the document to build the window to display its data. For examples of the kinds of information that go into BuildWindow, see Section 3.4.4.

Building a window involves instantiating some window objects, initializing them, perhaps creating other objects to go inside the window objects, and so on.

5.3.3. Managing Data

The main CDocument management tasks pertain to opening and closing documents, updating/saving their data, and printing their data.

Each document keeps track of the state of its data, that is, whether the data needs to be saved. The *save* state of a document is typically reflected in the File menu, which allows the user to save or not save a document's data. Not all documents need this feature. Some documents contain data that is read-only or that never changes. `CDocument` has several methods for finding out whether a document's data needs to be saved and for setting the "needs saving" state of the data to `TRUE` or `FALSE`. When you are defining your own document classes, you must also define whether the data contained in a certain type of document needs to be saved and, if so, how the data can be saved, either to a file or a database.

In addition to defining how data can be saved, you must define a mechanism for setting up printing of the data. For each document type, you must define how to set up the printing page.

See Also: For details on how to set up printing, see the section on `CPrintMgr` in the *XVT-Power++ Reference*.

XVT-Power++ provides some default data management mechanisms that you may want to use or override in your particular documents:

- When you create a new document by calling `DoNew`, the default `DoNew` mechanism simply generates a `BuildWindow` message to that same document.
- When you open a document by calling `DoOpen`, the default `DoOpen` mechanism first brings up a dialog window that prompts the user for the name of a file to open. `DoOpen` then fills in the specified document's file pointer and calls `BuildWindow`. The user has the option of cancelling the *open* operation rather than entering a filename, in which case `BuildWindow` is not called. If you override the `DoOpen` method, perhaps to do some extra things, but still use these default mechanisms to find out filenames, then the overridden method should call the inherited `CDocument` `DoOpen` method.
- When you close a document, the default `DoClose` mechanism goes through all of the document's open windows and notifies them to close. When notified to close, each window checks its document's data to see if it needs to be saved. If it does, a dialog box appears that prompts the user to save the data, close without saving, or cancel the close operation completely. If the user chooses to save the data, the appropriate save message is sent to the document. If the user chooses to discard the data, the window closes. If the user

opts to cancel, then the window notifies the document that it did not close. That is, the window returns a value of FALSE. When this happens, the document stops the closing operation for all of its windows.

- When you save a document by calling DoSave, the default mechanism is for the document to verify whether it has a defined filename to which it should save the data. Each document has an XVT file pointer for storing the filename. It is a protected data member that derived documents can set. If you are not dealing with files in your system, you will need to override this default behavior. If there is a filename for the document, then DoSave simply returns a value of TRUE. If no filename has been defined for the document, then the document calls DoSaveAs. It is up to you to override DoSave and put in the actual code for saving the data and then calling the inherited CDocument DoSave to find out whether the filename has been defined. For an example of how to do this, see Section 3.3.3.
- There is also a default DoSaveAs mechanism that invokes a dialog box prompting the user for a filename to which it should save the data. Then it sets the specified filename and calls DoSave to do the actual saving.
- When you call DoPrint to notify a document that you want to print it, the document goes through all of its views and sends them print messages. In response, the views redraw themselves for the printer rather than for the screen. When the user selects "Print" from the File menu, these actions are performed automatically. You do not have to tie the fact that someone selects "Print" to having a document print. The same is true for page setup. If you are working on a platform that allows you to set up a page before you send it to print, then DoPageSetUp invokes the appropriate application-specific dialog box for setting up the printing page.

Note: Keep in mind that the default mechanisms of CDocument work in conjunction with those of CApplication, discussed in Section 4.3, which are called automatically when different menu actions are selected. For example, when the user selects "Open," "New," "Save," and so on, from the menubar, the application object automatically takes care of these operations, channeling them through the DoOpen, DoSave, DoNew methods and so on, to CDocument.

5.3.4. Managing Windows

Earlier, we stated that `BuildWindow` must be defined by each derived document class in order to specify what should be done to build a window. As more windows are created for the same document, you will want to take advantage of `CDocument`'s window management functions. If you need to find a particular window, `CDocument` has a `FindWindow` method that takes the ID number of a window and returns a pointer to that window. Another method, `GetNumWindows`, returns the number of windows associated with a given `CDocument` object. The `CloseAll` method closes all of a document's windows, regardless of whether their data has been saved—in contrast to the `DoClose` method, which halts if a window returns `FALSE` to the closing operation because the user cancelled the operation, data has not been saved, or there is some other impediment.

See Also: For details on `CDocument`, refer to the `CDocument` section in the *XVT-Power++ Reference*.

5.4. Chapter Summary

We have described how data is managed in XVT-Power++: how new documents are created, how documents are opened, how the data is saved and printed, and how documents are closed. We have noted the different dialog boxes that are provided by default at the document level, and finally, considered how a document manages the viewing of data. Now that you know how XVT-Power++ accesses and manages data at the document level of its application framework, you are ready to learn how data is displayed and manipulated at the view level. This information is presented in Chapter 6.

6

VIEWS AND SUBVIEWS

6.1. Introduction

The *view* resides at the third level of the XVT-Power++ application framework, serving as the layer that permits the programmer to display information on the screen. Views not only display information, such as a message or a drawing, but may allow the user to interact with the application. For example, a user not only sees a button but also can interact with it by clicking the mouse on it. Other views can interact with the user through the keyboard, displaying typed text. Still other views allow the user to make selections from a list of choices. As the user interacts with views, their differing states may be represented visually. For instance, a set of check boxes draws differently once the user has selected a choice. Thus, views display textual and graphical data, allow the user to interact with the application, and reflect the state of the application.

`CView` and its derived classes compose by far the largest branch of the XVT-Power++ class hierarchy and are the basis for much of the power and efficiency of XVT-Power++. Together, these classes give you access to a model for visual display that is functionally complex, yet easy to use, if you understand the model.

This chapter tells you what you need to know about views and how they work in XVT-Power++. You will learn about the tasks that views handle, the characteristics of views, some useful categories within the view object hierarchy, the XVT-Power++ coordinate system, use of the mouse, and how messages are propagated among certain kinds of views.

6.2. Tasks Handled at the View Level

Views display a representation of different kinds of data in an application. When a user reads in a file, the application opens a new document that represents the file. A view within a window associated with the document—say, an `NScrollTextobject`—displays the contents of the file on the screen.

If you want to display a hierarchical picture of the directory structure on your computer—as is typically done in browsers in different graphical managers—you can draw graphical objects such as icons, lines, and different colored shapes to construct a picture of the tree hierarchy. All of these graphical objects can send and receive events, which are actions that occur within the system. Whenever you move the mouse, click on a button, or release the mouse over a button, a view receives the message and may respond to it. Some views may ignore a certain message while others react to it.

In addition to mouse clicks, a view may respond to a keyboard event, a sizing event, or a move event. As an application runs, views may draw and redraw at different points. For example, when the user clicks on the border of a window to bring it to the front of the window stack, the part of the window that was covered and all of the views it contains (such as a scroller, buttons, and a text box) must redraw themselves. When the user clicks on the scroller, it responds to the scroll event by scrolling upwards/downwards or left/right. If the user selects a print option, sending a print message to the window and all of its views, they may respond by drawing themselves somewhat differently so they can be drawn on a printer rather than on the screen.

6.3. General Characteristics of Views

The primary characteristic of all views is that they draw themselves on the screen. Each view must supply its own drawing mechanism, which is done through a method called `Draw`. This method takes a constant `CRect` reference that indicates the clipping region for the drawing and performs all the operations necessary for drawing. XVT-Power++ takes care of the clipping automatically, so you usually ignore the clipping region.

```
void CLine::Draw(const CRect& theClippingRegion)
{
    CShape::Draw(theClippingRegion);
    win_move_to(GetCWindow()->GetXVTWindow(),
        *itsStartPoint + itsOrigin);
    win_draw_aline(GetCWindow()->GetXVTWindow(),
        *itsEndPoint + itsOrigin,
        itHasStartArrow, itHasEndArrow);
}
```

6.3.1. Drawing

If you examine the `Draw` method for different types of views, you will notice the native XVT Portability Toolkit calls for XVT's functions for drawing such objects as icons, arcs, lines, and so on, as in the following example:

```
win_draw_rect(GetCWindow()->GetXVTWindow(),&rct);
...
win_draw_oval(GetCWindow()->GetXVTWindow(),&rct);
...
win_draw_icon(GetCWindow()->GetXVTWindow(),
        HPhysical(itsFrame.Left() + itsOrigin.H()) + itsCenter,
        VPhysical(itsFrame.Top() + itsOrigin.V()),
        itsCurrentRID);
```

Some views can be built up out of other views, as discussed later in this chapter when we consider the `CSubview` class. A view can actually contain several other views, as is the case with a list box, which is a composite of a scroller, a grid, and several text objects. The outermost view (the list box object itself) may do very little of the drawing and allow the inner views to finish everything else. More specifically, the list box object draws the border area of the list box but does not draw the text items contained within it. There is no `DrawText` function called inside of the `CListBox Draw` method. Instead, the text views inside the list box draw the text items. In short, the drawing can either be done by the enclosing view itself, or part of it can be done by the view and the other part by any other view inside it.

6.3.2. Showing and Hiding

Related to a view's capacity to draw itself is its ability to show and hide itself. There are times when you may want to tell a view not to draw itself anymore by sending it a `Hide` message as shown here.

```
aView->Hide();
```

When you send a view a `Hide` message, you are notifying it not to respond to update events from that point on. You are not notifying it to change the way the screen looks by immediately becoming invisible. You are just changing its behavior. You will at least have

to send a Draw method to the enclosure of the view you are hiding and give it a clipping region big enough to cover the view.

```
aView->Hide();
aView->GetEnclosure()->DoDraw(aView->GetFrame());
```

You may wonder why, when you tell a view to hide, you don't just redraw the region so that this view is no longer visible on the screen. The reason is that sometimes you do not want to hide just one view; you may want to hide three or four or one hundred views at a time—without the flashing that would occur if every single view that received a Hide message also got an update event to redraw its part of the screen. Instead, you notify all views involved to hide themselves (or perhaps change their state in some other way by sending a different message). Once you are done, you send one update event that takes care of redrawing an entire region that is now in a new state. The same applies when you tell a view to show itself if it was previously hidden. You are notifying the view to respond to update events and draw itself *from now* on. This does not take effect until the next update event.

6.3.3. Activating and Deactivating

In addition to changing the visibility of views through Show, Hide, and Draw messages, you can notify views to be active or inactive. For example, a window may contain a spreadsheet with a grid full of text fields that can be activated one at a time so a user can type something into each field. An active view is currently receiving keyboard events. This does not mean that other views are disabled, just that they are not active. You can activate another text field simply by clicking on it. Now when you type, all of those events will go to that particular view.

Any view can receive an Activate or Deactivate message. Some views have special definitions of what it means to be active or inactive. It is up to you to override the Activate message for a particular view that you are designing and specify its behavior when active. You can set a view so that it becomes active when a user selects it for dragging. You may want to deactivate it later.

Windows, like other views, can be active or inactive. A window that is at the front of the window stack and is receiving events is the active window. Any window behind it is not active at the moment.

6.3.4. Enabling and Disabling

Enabling or disabling views is different from making views active or inactive. Disabling a view notifies it not to respond to any events from the user, i.e., keyboard or mouse events. If a user clicks on a disabled view, it will not respond. However, the view will still respond to certain other events. For example, an update event for a view to redraw itself is not affected by the fact that the view is disabled. If you want to enable the view again, you must send it an `Enable` message.

```
aView->Enable();
```

A disabled view will not accept an event from the mouse or keyboard, and in addition, it may look different when it is disabled. This is commonly true of icons. In many systems, the look of a view when it is disabled is fuzzy or grayed out.

6.3.5. Dragging and Sizing

Every view has a certain size and location, which can be changed when the user drags it around. You can set the dragging or sizing properties of a view to be on or off. You can also set a view to be automatically selected when the user clicks on it so that the user can drag the view to move it or size it, as is done in many drawing programs. This behavior is achieved through the `CWireFrame` class, which is a friend class that can respond to sizing and dragging mouse events.

Views automatically take care of any scrolling that has to be done on the screen. Some views act as enclosures for other views and can scroll their contents when the user manipulates a scrollbar (see Section 6.5.3).

6.3.6. Setting the Environment

Another characteristic of views is that they each have access to a helper environment object that specifies a view's color and the width, pattern, and color of the pen used to draw the view. Also specified is the color and pattern of the brush used for painting the inside of the view. The environment specifies the fonts as well. If the environment changes, it signals to the view to change the way it draws itself. A view may have to change its size if the font has changed or it may change the width or height of its borders, depending on the width of the pen.

```
void CText::SetFont(const FONT& theFont, BOOLEAN isUpdate)
{
    CView::SetFont(theFont, isUpdate);
    RecalculateSize();
}
```

6.4. The View Object Hierarchy: Enclosures and Owners

This section considers the possible relationships between the various view objects in the XVT-Power++ system.

6.4.1. Enclosures and Nested Views

Earlier, we alluded to the fact that some views can be built up by putting several views together and inserting them into a larger view that is capable of containing different kinds of smaller views. This composition is made possible by a relationship between an *enclosure* and a *nested* view. An *enclosure* view is a view that contains one or more views. The views inside the enclosure are *nested* views. Note that every view must have an enclosure. A text view may be nested inside a list box, which in turn is nested inside a window. In other words, the window is the enclosure of the list box, and the list box is the enclosure of the text view.

Enclosures ensure that every view contained inside them is clipped to them. *Clipping* means that a view contained inside an enclosure cannot draw itself beyond the boundaries of its enclosure. Thus, a nested view may be only partially visible. For example, a text object or a picture contained within a window clips to the window's border so that any text or part of the picture extending beyond the window's border is not visible and will need to be scrolled or otherwise moved in order to become visible within the window's borders. Similarly, when you set different enclosures inside a window, whatever is nested is also clipped to the border of its enclosure.

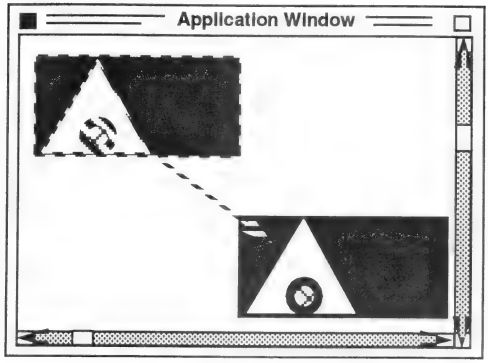
Windows are a special case because a window cannot be enclosed inside another view. They still have a logical enclosure, which is the screen, or, on some platforms such as Windows and Presentation Manager, a "task window." The enclosure of a window is something over which you have no control.

When a view other than a window is created, XVT-Power++ needs to know what enclosure to give that view. Thus, the constructors of most views have a parameter that must pass a pointer to a view that is acting as the enclosure.

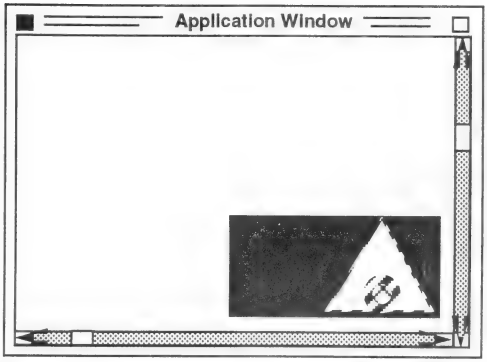
```
CScroller* aScroller = new CScroller(someWindow,
    CRect(0,0,100,100));
CIcon* anIcon = new CIcon(aScroller, CRect(10,10, 42,42));
```

All views nested inside an enclosure are drawn relative to the origin of that enclosure. Each enclosure in effect sets up its own coordinate system, and the views it contains draw within that coordinate system. If the entire enclosure is moved to a new location, the views nested in the enclosure are oblivious to the move, continuing to draw in coordinates that are relative to enclosure. In other words, the enclosure defines space for a nested view, and the larger context of the screen is irrelevant.

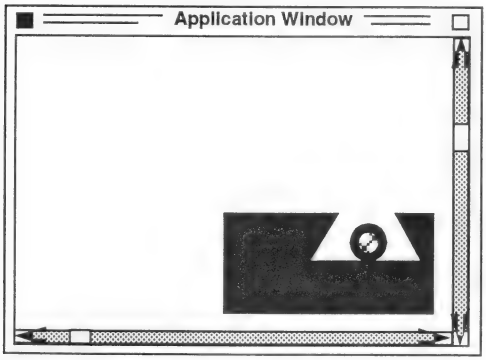
While some types of nested views can act as enclosures, it is not true that *any* view can enclose another view. There is a reserved, well-defined set of views that cannot act as enclosures. The top-level view class is `CView`, which defines all the properties of views that have been discussed so far. From `CView` a class called `CSubview` has been derived. Only views that inherit from `CSubview` can act as enclosures; views that inherit directly from `CView` *cannot* act as enclosures. In Figure 11, the views inheriting directly from `CView` appear in the gray area.



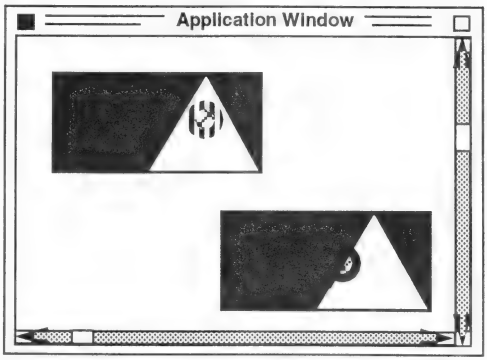
To illustrate the behavior of nested views, this rectangle contains a triangle that in turn contains a circle. When the rectangle is moved, all of the objects nested within it move with it.



When the user clicks on the triangle to move it, the triangle can move freely inside the rectangle object.



Each shape object is clipped to its enclosure so that if a part of an enclosed object extends beyond the clipping region, that part is obscured—as the top of the triangle is here.



The circle can move freely inside the enclosing triangle and becomes partially obscured as it extends beyond its clipping region.

Figure 10.Nesting Behavior of Views

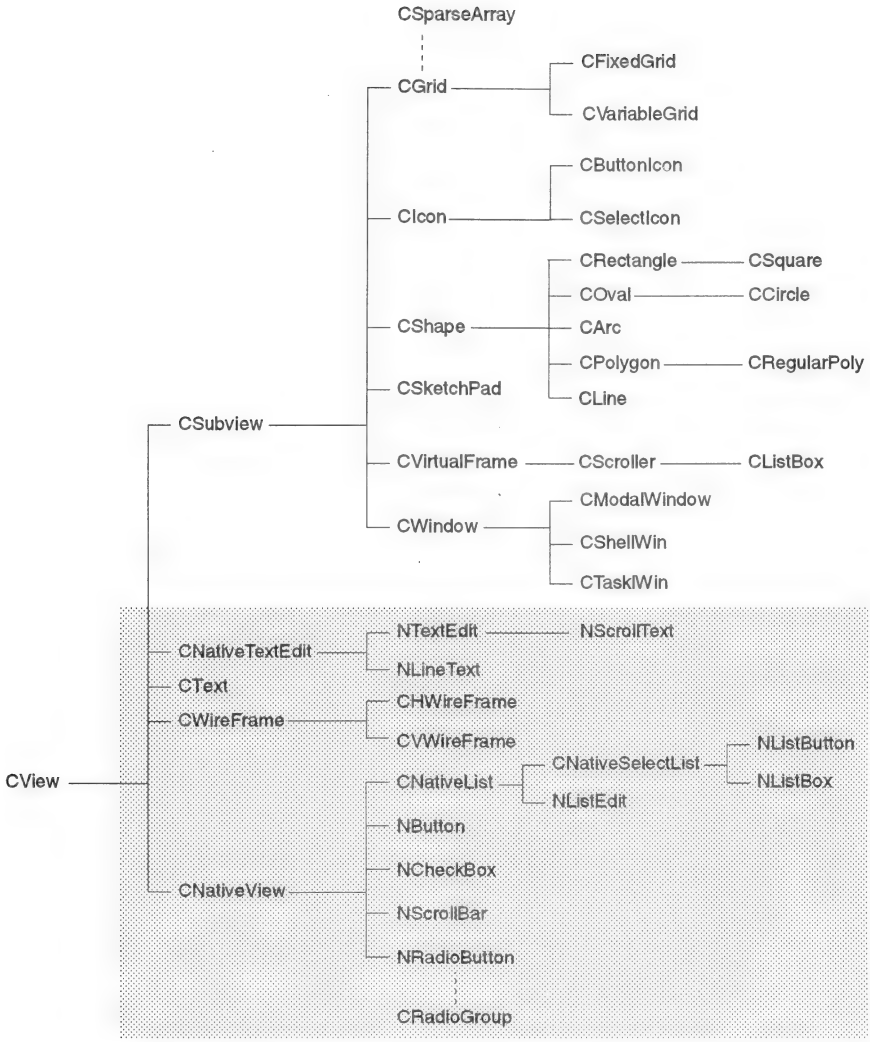


Figure 11. XVT-Power++ View Hierarchy

As Figure 11 shows, a `CText` object is a type of object that does not inherit from `CSubview`. `CText` is a view that allows you to display static text on the screen, for example, "Enter Password:" on a login window. It seems fairly obvious why this text object cannot function as the enclosure of something else. Another glance at the XVT-Power++ hierarchy tree reveals other `CView` classes that

cannot act as enclosures, such as `CNativeView`. Native views are objects that are drawn by the window manager, not by XVT-Power++ and not by the user. Since XVT-Power++ does not draw them, XVT-Power++ cannot draw anything else inside them either. While they can be displayed inside other views, nothing else can be displayed inside them.

On the other hand, `CGrid` is a type of view that inherits through `CSubview` and therefore can act as an enclosure. You can nest many different kinds of views inside of a grid, inserting the items in its rows and columns.

6.4.2. Owners and Helpers

Thus far, in looking at the XVT-Power++ object hierarchy, we have discussed one possible relationship between view objects, that of enclosures and nested views. There is another relationship between view objects that needs to be considered: ownership. Different views can be owners of other views that act as helper or auxiliary objects and provide a service to the owner view. If a view object that owns a helper object is closed, the owned object is also closed. If a view is destroyed, any objects that it owns are also destroyed.

6.4.2.1. CGlue

One example of a helper object is a `CGlue` object, which provides the "stickiness properties" of its owner. *Stickiness* refers to the behavior of a view when its enclosing view is sized. Depending on its type of stickiness, a nested view will stretch with its enclosure or stay fixed by a constant distance from the borders of its enclosure. Suppose you want a rectangle to be "stuck" to all four sides of its enclosure, which is a window. You would give it a glue type of `ALLSTICKY`, as follows:

```
aRectangle->SetGlue(ALLSTICKY);
```

When the window stretches during resizing, the rectangle also stretches. When the window shrinks, the rectangle does, too. If you had specified a "bottom right" type of stickiness, only the bottom and right sides of the rectangle would be stuck to the window. In this case, if the window were resized, the rectangle would move along with the right bottom corner of its enclosing window.

See Also: For a listing of all the types of stickiness that can be set for a view, refer to the section on `CGlue` in the *XVT-Power++Reference*.

Instead of embedding all the code inside CView that tells it how to act as a "sticky" object, we have defined a separate CGlue object to take care of the logic of stickiness. When you specify the stickiness of a view, a CGlue object is automatically created, and it knows that its owner is the particular view that you have made sticky. Thus, there is an "owner" relationship in which the view owns the glue.

6.4.2.2. CEnvironment

Another helper class is CEnvironment, which provides an object that keeps track of its owner's colors, pen, brush, fonts, drawing mode, and so on. You set the environment of a view as follows:

```
CEnvironment env(COLOR_BLUE, COLOR_RED, COLOR_WHITE,
    PAT_SOLID, COLOR_BLACK, PAT_SOLID, 1, STDFONT,
    M_COPY);

aView->SetEnvironment(env);
```

When you set the environment of a view, that view creates its own environment object to store this data. When the owner view is destroyed, its environment object is also destroyed.

An environment object can be shared by many views. For the sake of a consistent look-and-feel, an application typically has several windows and other views that use the same environment. Thus, borders are drawn in the same color, the brush patterns and colors in the view interiors are the same, the fonts appearing in text views are consistent, and so on. Although one particular view acts as the owner of an environment object that is destroyed when it is destroyed, other view objects may also be using that environment while it exists. This is possible because views can only share an environment with an object that is above them in the object hierarchy; that is, with an enclosure. The types of sharing relationships that can be established between owners of environment objects is discussed in Section 7.7.

6.4.2.3. CWireFrame

CWireFrame objects are yet another class of helper objects that can be owned by other views. A wire frame is an object that enables a view to be moved and sized. On the screen, a wire frame appears as a rubberband (flexible) frame surrounding an object that is being sized or dragged. You set the sizing and dragging of an object as follows:

```
aView->SetDragging(TRUE);

aView->SetSizing(TRUE);
```

When you thus specify that a view object is to be sizeable and draggable, that view creates its own wire frame object. At the appropriate times, the wire frame can take care of different events that it may receive, such as mouse downs and mouse dragging. The wire frame also can notify its owner of the new location to which it is being moved or of the new size to which it is being stretched.

6.4.2.4. CPoint and CRect

Each view has its own `CPoint` and `CRect` objects that enable it to keep track of where it is on the screen; that is, of the region where it is located, the origin from which it is drawing, and so on. When the owning view is destroyed, its `CPoint` or `CRect` object is destroyed as well. These two heavily used helper classes are discussed in detail in Section 6.5.

6.4.3. Similarity Between Enclosures and Owners

While the distinction between enclosures and owners is real and useful, there is an important similarity between them that contributes greatly to XVT-Power++'s ease of use. To an extent, the relationship between an enclosure and the views nested within it is also an owner/helper relationship. We noted earlier that when an owner view is destroyed and deleted, so are all of its helper objects. Similarly, when an enclosure is destroyed and deleted, so are any views nested inside it. Moreover, if the views contained within the window also contain nested views of their own, these views will also conveniently close. This means that you have to take care of very little memory management. You can simply create views as desired, knowing that if at a certain point you close a window, the window will take care of closing anything inside it.

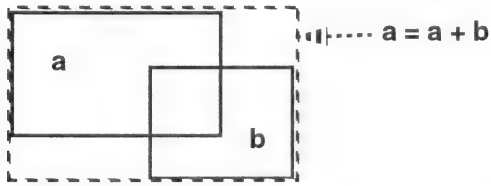
6.5. The Coordinate System

Coordinate systems are crucial to the use of views because the screen location of a view must be specified before it can be drawn. Success in using XVT-Power++ depends in many ways on understanding the coordinate system. You must know about screen-relative, global (window-relative) and local (view-relative) coordinates and when these coordinates are applied in order to use them correctly and manage them efficiently. The basis for managing the coordinate system is provided by XVT-Power++'s `CRect` and `CPoint` classes. This section discusses these classes and the different types of coordinates in detail.

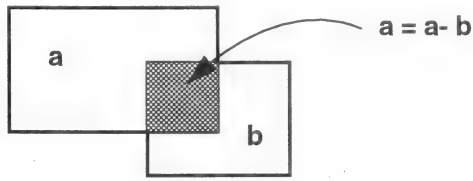
6.5.1. CRect

In an abstract sense, `CRect` is simply a class that provides a data structure for storing information about a rectangular region of the computer screen. `CRect` manages four different values: left,top, right, and bottom. Together, these four values form a rectangle that is located somewhere in space. Each of the four values represents a value in the coordinate system. The `CRect` object itself has no information about what coordinate system it is mapping. It merely contains the mapping.

`CRect`'s mapping allows you not only to set different rectangular regions but also to perform many useful operations, such as taking the union of some rectangular regions. A union operation adds one region to another, which has the effect of resizing the region. Suppose you want to update the entire region that encompasses two overlapping views. You can take the union of one view's rectangular region with the other view's rectangular region, which has the effect of an addition. The result is a much larger region that contains both of the regions.



Similarly, you can take the intersection of two views, which has the effect of a subtraction. For example, if a view is partially covered by another view or is being clipped by an enclosure, you can take the intersection of those two views through the `CRect` to find the shared rectangular region.



In addition to union and intersection, you can perform several other operations on a region . You can expand it or shrink it, and you translate it from one position on the screen to another. All of these

operations for managing regions on the screen are available through the `CRect` class.

See Also: For details on `CRect`, see the section on this class in the *XVT-Power++ Reference*.

6.5.2. CPoint

`CPoint` is similar to `CRect` in that it also manages positions within XVT-Power++'s coordinate system. However, `CPoint` manages a value for a single point or single unit. Instead of an entire rectangular region, `CPoint` consists of a horizontal and a vertical value for one position in space. It is irrelevant to `CPoint` where the coordinate system is (that is, whether it originates at the top-left of the screen) or what the sizes of the units are (whether in pixels, characters, inches, or some other measure).

`CPoint` allows you to move from one point to another, set either the horizontal or vertical coordinate of a point, add points together, subtract one point from another, or make two points equal. Moreover, it includes some methods for coordinate system conversion so that you can translate the coordinates of a point, covert a `CPoint`'s coordinates from view-relative to window-relative coordinates, and vice versa.

See Also: For details on `CPoint`, see the section on this class in the *XVT-Power++ Reference*.

6.5.3. The Point of Origin

Central to any coordinate is the point of origin from which the coordinate is calculated. That is, before a coordinate such as *3,9* can make any sense, we need to know the context for the numbers—whether the point is screen-relative, window-relative, or view-relative. In XVT-Power++, the need to know the context of a coordinate is complicated by the fact that each enclosure defines its own coordinate system. In short, the context is crucial.

Each XVT-Power++ view uses a point of origin to calculate its position on the screen. This origin is the *0,0* point that normally represents the position of the enclosure's top-left corner. Every view knows where its enclosure's top-left corner is because it has a `CPoint` object it uses called `itsOrigin`. The origin is automatically managed and updated in XVT-Power++. If the enclosure changes from one position to another, the origin of any nested views is updated automatically and internally. There are some methods for

manipulating a view's origin, but these are reserved for advanced XVT-Power++ programming. Normally, there is no reason for you to concern yourself with a view's origin.

Suppose your application contains a scroller in which a circle is nested. Typically, the circle draws at a certain position relative to the scroller's top-left corner. However, when a user clicks on the scrollbar, the circle draws at a different position because the entire contents of the scroller have been scrolled to the left or right, up or down, depending on the scrollbar's orientation and the direction of the click. Now the origin for the circle becomes the top-left corner of the scroller, plus or minus some scrolling coefficient. It is usually unnecessary for you to know these specifics of how the point of origin changes with a scroller. We mention it here to illustrate the importance of origins and how they are used to keep track of where views should draw relative to an enclosure, even if the enclosure happens to be in a particular state, such as a scrolled state.

6.5.3.1. Screen-Relative Coordinates

Windows are the outermost enclosures. All windows are placed relative to the screen's top-left corner and are the only screen-relative views in XVT-Power++.

6.5.3.2. Global, Window-Relative Coordinates

In many places in XVT-Power++, you'll encounter the term "global coordinates." This term is synonymous with "window-relative coordinates"—"global coordinates", as it is used here, refers to the window, not to the entire screen.

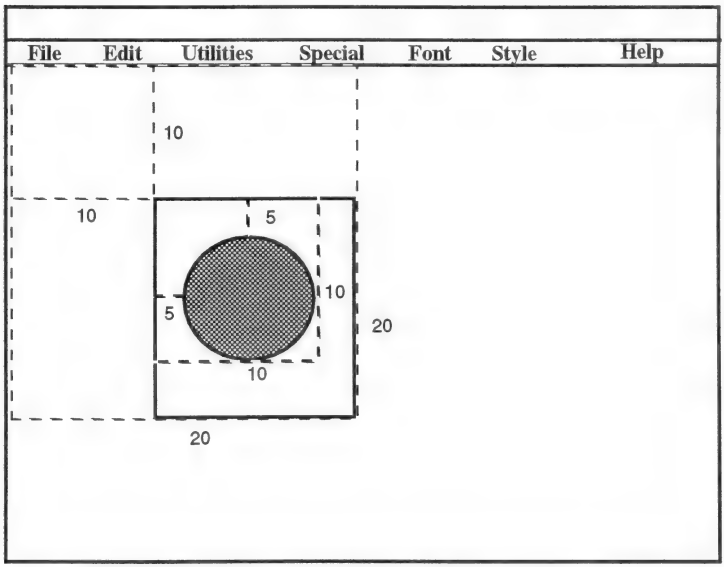
Several types of methods take window-relative coordinates. For example, the "`Do-`" methods, such as `DoMouse` and `DoDraw`, all take global (window-relative) coordinates, as do all the drawing methods for any region these methods receive. More precisely, they take a region or a certain point where the mouse is clicked, which is relative to the entire window. Other methods, such as the basic mouse methods without the "`Do-`" prefix, take view-relative coordinates. Thus, when a view gets a `MouseDown` message and the mouse is at point *5,5,* this point is relative to the view's top-left corner. It is important to look at a method and see what kind of coordinates it takes.

Suppose that you want to insert a view, such as a rectangle, into a window. You place the top-left corner of the rectangle at coordinate *10,10* and the bottom right corner at *20,20*. These coordinates are relative to the window's top-left corner because the window, as an

enclosure, sets up its own coordinate system, and anything that is inserted directly inside of it is positioned relative to its top-left corner.

6.5.3.3. Local (View-Relative) Coordinates

Suppose you decide to place a view inside the rectangle, say, a circle. The circle is inserted at position *5,5* and extends to *10,10*. These coordinates, as you may have guessed, are relative to the *rectangle*'s top-left corner and are thus view-relative coordinates.



It is important to understand this as you use different methods in XVT-Power++ that take a region. This is because it is indicated very specifically whether the coordinates of the region are relative to the screen, a window, or a certain view.

6.5.4. Units of Measure

Another critical issue for coordinate systems besides the point of origin is the size of the units—whether they are pixels, inches, centimeters, characters, or a user-defined unit. If the units are pixels, each unit maps one-to-one with pixels on the screen. If the units are pixels and a `CLine` object, for example, extends from point 0 to point 10, then ten pixels on the screen are turned a certain color to represent the line.

By default, XVT-Power++ uses pixel coordinates. However, to maximize the portability of an application, as well as the ease of programming a graphical user interface, you often may not want to use pixels because they provide device-dependent coordinates. An application that looks beautiful running on one platform may not look right any more when it runs on a different machine with a smaller screen or completely different resolution. When you combine text and graphics, you typically position the graphics and shape/scale them in terms of the text. If the application runs on a different machine where the font size is totally different, suddenly the graphics will be "off."

The solution to this problem is to use logical coordinates rather than physical or device-dependent coordinates when you program a graphical interface. Through the `CUnits` class, XVT-Power++ allows you to use a number of logical coordinate systems. `CUnits` allows you to program in coordinates that map to inches, centimeters, a user-defined unit that maps logical coordinates to some physical representation on the screen, or character units in a font that you choose (normally the system font). You can specify, for example, that you want a two-inch by three-inch rectangle or an object that is ten characters long and three characters wide.

6.5.5. Translating Coordinates

XVT-Power++ provides facilities for translating from one coordinate system to another. Suppose you have defined a view that inherits from `CSubview` and therefore can act as an enclosure. This view contains several nested views. Moreover, it is defined to trap and receive any mouse events that occur within its region, regardless of whether an event occurs within the region of one of its nested views. Now suppose that this enclosure gets a `MouseDown` message, which, in XVT-Power++, has a `CPoint` coordinate that is relative to the view receiving the message. This point indicates where the mouse happened to be, in this case, at point *5,5*. The view sends the message down to another view that is nested within it at this location by calling the nested view's `MouseDown` method, which takes as its first parameter a `CPoint` coordinate that is relative to the nested view's coordinate system.

The point that the enclosure received is in coordinates relative to the enclosure itself. Thus, the enclosure must translate the point into coordinates that the enclosed view can use. This is a case where we want to translate a point from one view's coordinate system (the enclosure) to another view's coordinate system (the nested view). The `CRect` and `CPoint` classes provide several easy-to-use utility

methods that do the translation for you. There are methods for localizing and globalizing different points. In the case considered here, you simply call the `CPoint Translate` method, which takes two parameters: a view from which to translate the coordinate and a view to which to translate the coordinate. You call `Translate` and pass it the enclosure and the nested view, as follows:

```
aPoint.Translate(theEnclsure, theNestedView);
```

6.6. The Mouse

One of the key features of graphical user interface applications is that they are mouse-driven. The mouse is a primary means by which the user interacts with the application. This section considers the kinds of interactions that occur as views receive mouse events.

6.6.1. The Basic Mouse Methods

First. let's look at the different kinds of mouse events that can occur in XVT-Power++:

- `MouseMove` — Generated as the mouse cursor moves around on the screen, independently of whether a mouse button is being pressed
- `MouseDown` — Generated when the user presses down a mouse button. Since a mouse can have from one to three buttons, each button can have a different meaning when it is pressed in an application. Thus, different values are assigned to the `mouseDown` event, depending on which button is pressed
- `MouseUp` — Generated when the user releases a mouse button and depends upon the value that has been assigned to that particular button
- `MouseDouble` — Generated when the user double clicks a mouse button; a `mouseDouble` event has a different meaning than just clicking the button once

The view classes can accept any of these four types of events. When it is determined that a particular view is the target of a mouse event, one of these four methods is called and the view receives a message. Each of these methods has three parameters:

- A parameter that contains a point, called `theLocation`, that indicates exactly where the mouse was when the event occurred. This location is a point representing, in logical units, the coordinate relative to the view containing the point

- A parameter indicating which mouse button, if any, is associated with the event. This parameter takes a value of 0, 1, or 2 (with 0 representing the leftmost button), depending on which button was pressed. For a one-button mouse, the button value will always be 0
- A parameter that indicates whether the Shift or Control key was pressed in conjunction with the mouse button

Most view classes are programmed to respond with very specific behaviors to mouse events. For example, all moving and dragging is handled by the views themselves. A button is an example of a view that is programmed to respond to `MouseDown` and `MouseUp` events in a certain way. When a user presses a mouse button over a button view, the appearance of the button view changes to indicate that it has been activated. Upon a release of the mouse button, the button view's appearance changes again, and the `MouseUp` event generates a `DoCommand` message.

Typically, mouse events occur in a sequence. By default, all `CView` objects generate a click command whenever the mouse is clicked on them. A mouse "click" is a sequence of a `MouseDown` followed by some possible mouse moves and concluded with a `MouseUp`. In other words, to "click" the mouse, the user must at the very minimum press and release the mouse button, generating a `MouseDown`, `MouseUp` sequence. If the user presses down a mouse button over a view and then drags the mouse outside the view's region without letting go, this is not a click. Such behavior enables a user to prevent an unintended click by simply dragging the mouse away from a view before releasing the mouse button.

6.6.2. The "Do-" Mouse Methods

The mouse events just described are not the only ones included in the interface. Actually, there are eight events.

- `DoMouseDown, MouseDown`
- `DoMouseMove, MouseMove`
- `DoMouseDouble, MouseDouble`
- `DoMouseUp, MouseUp`

The "`Do-`" mouse methods are the methods that trigger the mouse method calls. They are decision-making methods in that they determine which view should receive a given method. Normally, a view receives a "`Do-`" mouse event and a "`Do-`" mouse method is called for that view. The "`Do-`" mouse methods carry the same kind

of information as the basic mouse methods—a `CPoint` location, a button value, and an indication of whether the Shift or Control key was also pressed. However, the `CPoint` that indicates where the mouse event occurred is in *window-relative* coordinates because it has not yet been decided which view should receive the event. The only thing known at this point is the particular window that should be in charge of this method.

Inside every "`Do-`" mouse method, the following 3 steps occur:

1. It determines which view should receive the event.
2. It localizes the location to the coordinate system of the view that is to receive the event.
3. It calls the receiving view's basic mouse method `theLocation` with the localized information.

The mechanism in XVT-Power++ for deciding which view should receive a mouse event is the concept of the "deepest subview." When a mouse event occurs at a certain location, that event first goes to the window. The window then searches for the deepest view that contains this location. "Deepest" means that several views can be nested around a given point and that the target of the mouse event is the view that is nested most deeply. Finding the deepest subview is accomplished through the `CView FindEventTarget` method. For classes derived from `CView`, you can override this method, perhaps changing the mechanism for finding the deepest subview so that it returns a different target for the mouse event. `FindEventTarget` is commonly overridden for views that handle an event themselves instead of passing it on to another view.

If the view that is to receive the event is movable and/or sizeable, this state is treated as a special case. For example, if a button receives a mouse event, it typically changes its appearance when pressed and generates a `DoCommand` when it is released. However, if the button is movable and sizeable, the normal behavior does not occur when the user clicks on it. Instead, a rubberband frame appears around the button so that it can be dragged to another location or sized. XVT-Power++ must therefore be able to find out whether the view that is to receive the event is movable or sizeable. This is achieved through a method called `FindHitView`. Once `FindEventTarget` returns a view as the target of the event, then `FindHitView` is called on that view. `FindHitView` either returns the view itself or the view's wire frame. As noted earlier, `CWireFrame` is the class that implements all the moving and sizing in

XVT-Power++. When a view is movable or sizeable, it channels the mouse events it receives to its wire frame helper.

Following is a summary of the steps that occur in sending a mouse event to a view:

1. A user clicks the mouse over a certain point on the screen.
2. A `CSubview` "`Do-`" method receives this event and interprets the event's location in window-relative coordinates. It is clear which window should be in charge of the event.
3. The "`Do-`" method calls the `CView` `FindEventTarget` method to find the deepest subview containing the event's location.
4. `FindEventTarget` calls `CView`'s `FindHitView`, which returns either the target view or its wire frame.
5. If `FindEventTarget` returns the target view, the "`Do-`" method localizes the event location to the coordinate system of the target view.
6. The target view's basic mouse method is called with the localized information.

6.7. Subviews

Earlier, we observed that any XVT-Power++ view can, and in fact must, be nested inside an enclosing view, with the exception of the window, whose logical enclosure is the screen or the task window. However, not all views can act as enclosures. The subset of views that can act as enclosures is clearly defined by a class called `CSubview`. This section discusses in detail all of the views that can act as enclosures and the implications of this property within the larger context of XVT-Power++.

6.7.1. Nesting Behavior

"Nesting" means that an enclosure can contain numerous views that are nested at the same level, perhaps overlapping, as well as views that act as enclosures for nested views of their own.

Let's first look at the behavior of overlapping views. Suppose there is a window in which you have inserted a circle object. You decide to place a square directly on top of the circle. In this case, the square is not nested inside the circle. Both shapes are nested inside of their common enclosure, a window, but they are overlapping—or, more precisely, stacked. In a stack of views, the last view to be created and

inserted into the window is the top view at that location. In this case, the square is on top of the circle. If you click the mouse on the area shared by both the square and the circle, the square will receive the event because it is covering the circle, and it is as if the circle is not even there.

XVT-Power++ provides several ways to control which view is on top of a stack of views. One way is through the order in which the views are created since the last view created is the one on top. Also, `CSubview` has two methods, `PlaceTopSubview` and `PlaceBottomSubview`, for placing a certain view at the top or bottom of a stack. In the example discussed here, the square lays directly on top of the circle and is the top subview of the window, while the circle is the bottom subview.Say you decide that you want to reverse this order by placing the square beneath the circle. You call the window's `PlaceBottomSubview` method, which it inherits from `CSubview`, and give it the square as a parameter. The square then becomes the bottom subview in the window. `PlaceTopSubview`, as you can imagine, works very much the same way.

The interface of `CSubview` allows you to find out several different things about a given enclosure's nested views. You can find a view that is nested inside the enclosure, either by using a view ID or by specifying a location in coordinates relative to the enclosure and getting the top-level view that contains this `CPoint`. You can also get a list of every view that shares this certain point. These operations are made possible by the `CSubview` `FindSubview` and `FindSubviews` methods.

There are times when it is desirable to circumvent `FindEventTarget` and always send the event to a particular view. This is especially true when you are dragging or sizing a view but want it to receive all mouse events. In this case, you call the `CSubview` `SetSelectedView` method on a given enclosure so that the enclosure will send all events to the specified view. If later you decide to take away the view's status as the selected view, you can call `SetSelectedView` again and give it a value of `NULL`. Then the enclosure will revert back to the default behavior of searching for a nested view via `FindEventTarget` when it receives a mouse event.

6.7.2. Propagating Messages from Enclosures to Nested Views

XVT-Power++ has many different types of messages that are propagated from an enclosure to all of its nested views, and then from each of these nested views (which may also be enclosures) to

their respective nested views, and so on in a recursive fashion all the way down to the deepest views. Suppose you have inserted into a window a rectangle that contains many different kinds of objects. Several of the objects enclosed inside the rectangle contain other objects inside of them. Now you want to send an update message that will reach all of these objects, from the rectangle enclosure to the tiniest and deepest enclosed view. You would send a `Draw` message that can propagate recursively as described here.

Typically, the messages sent to `CSubview` objects—such as drawing, showing/hiding, activating/deactivating, enabling/disabling, dragging, and sizing—consist of two methods: the base method and the "`Do-`" version of that method (i.e, `Draw`/`DoDraw`, `Show`/`DoShow`, or `Activate`/`DoActivate`). The "`Do-`" methods are in charge of the propagation scheme. Thus, when a base method such as `Draw` is called on an enclosure, the enclosure uses its own `Draw` method to draw itself and then propagates the message by calling all of the nested views' `DoDraw` methods. This means that each of the nested views will use its own `Draw` method to draw itself and then call the `DoDraw` methods of all of its nested views. In this way, the message propagates recursively until all views have received it.

6.7.3. The "Wide Interface"

This schema works quite well for all views that can act as enclosures, that is, for objects derived from `CSubview`. It is unnecessary for views that derive directly from `CView`. A `CView` object, has no need to propagate messages because it cannot contain nested views. Thus, when a view from a class deriving directly from `CView`—for example, `CNativeView`—receives a `Draw` event, it simply draws itself.

It is clear that only views inheriting from `CSubview`, that is, views that can act as enclosures, need and use the "`Do-`" methods. For example, you might expect `DoShow` to appear only at the `CSubview` level and not at the `CView` level in the XVT-Power++ class hierarchy. However, if you examine `CView`, you will notice that it contains the "`Do-`" methods as surely as `CSubview` does. For any `CView`, you can call `DoShow`, `DoDraw`, `DoHide`, and so on. XVT-Power++ is structured this way so that the `CView` and the `CSubview` classes can have a very compatible interface. In other words, when you get access to a view object, you do not have to worry about whether it has the specific properties of a `CView` or `CSubview` object. You can treat these objects in the same way.

At the `CView` level, a "wide interface" is in place to make `CView` objects almost identical in use to `CSubview` objects. `CView`'s "`Do-`" methods simply call the view's basic method and do not attempt to do looping through enclosed views because there are not any enclosed views. For example, `DoDraw` simply calls `Draw`. While this "wide interface" is redundant, it is intended as a convenient arrangement for you, the application developer. We refer to this "wide interface" at several points in the *XVT-Power++ Reference*.

6.8. Chapter Summary

Now that you know how XVT-Power++'s object hierarchy is organized for views and how messages are propagated within this hierarchy, you are ready to learn the details of the event and message passing structure within the larger context of XVT-Power++'s application framework. These details are presented in Chapter 7.

7

APPLICATION FRAMEWORK

7.1. Introduction

The preceding three chapters separately consider the `CApplication`, `CDocument`, and `CView` classes in the XVT-Power++ class hierarchy. This chapter describes how these three classes fit together to compose XVT-Power++'s application framework. It outlines the three levels of XVT-Power++'s application framework in terms of the basic tasks performed at each level, and then it explains how messages are propagated throughout this structure. The rest of the chapter focuses on aspects of the framework that are of concern to any XVT-Power++ application developer: setting up menus and handling menu commands, handling keyboard events, defining the look-and-feel of your application by setting display properties such as colors and fonts and drawing modes, and becoming familiar with the printing facility. The chapter concludes with a look at XVT-Power++'s shell classes as examples of how to derive new classes from `CApplication`, `CDocument`, and `CView`. It describes the Shell utility that is provided to help you get started.

7.2. Levels of the Framework

The purpose of an application framework is to provide a well-defined structure, and thus a common design, for every application developed around it. An application framework allows the developer to reuse a program design. Here, "reuse" goes beyond code reuse by which the developer reuses classes or extends them by deriving from them. If an application framework is generic enough to meet the needs of very different programs, then its entire design and structure is reusable. The XVT-Power++ application framework is designed to be reusable in such a way. This framework is built to accommodate the tasks that most graphical user interface

applications typically perform. It is structured to deal with these tasks on three different levels:

7.2.1. Flow of Control

There must be a logical order to the events that occur within the system, whatever they may be. The XVT-Power++ application framework assigns responsibility for the flow of control to its top level, where the `CApplication` class resides. For each application developed on XVT-Power++, this role is performed by a user-derived object of type `CApplication`, as discussed in Chapter 4. This object takes care of starting the application, initializing it, shutting it down, and cleaning up—in short, the overall topmost logic for execution.

7.2.2. Accessing and Managing Data

The second level of the XVT-Power++ application framework, the `CDocument` level, is in charge of any data that an application will be using, regardless of its type—whether it is graphical or textual data, and whether it is statically stored or dynamically created during execution and discarded when the program terminates.

7.2.3. Displaying Data

The third and final level of XVT-Power++'s application framework, the `CView` level, is responsible for the generic task of displaying data. The term "data" is a little misleading because it refers to all viewable features of an application. We do not normally think of a button on a window as having data associated with it. Rather, it has an associated action, as when you click on a button to cancel your selection of Quit from the File menu. XVT-Power++ uses views either to display data, as with a text file, or to interact with the user and then give information to the application, as with the button.

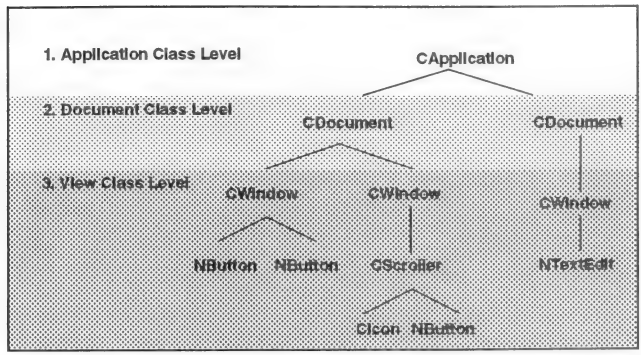


Figure 12. Application Framework for a Typical Application

7.3. Propagating Messages

The core of XVT-Power++'s application framework is the ability of its different levels to communicate with each other and to delegate tasks to each other. For example, the user interacts with the application through the view level and sees the running application in terms of the views that have been created on the screen. The user can send information to the application by interacting with those views through the mouse or keyboard. Some of the user input or some of what the application communicates to the user may be information that should be handled at a level other than the view level. There must be a defined communication scheme, event propagation scheme, or delegation scheme to make possible the communication between the user and the appropriate level of the application framework. XVT-Power++ has three main channels for message passing.

7.3.1. Bidirectional Chaining

Messages (typically XVT Portability Toolkit event messages) go from an event handler to some window to some particular subview. Once it is at that subview, the message may turn around and start propagating back up. Some events go to a selected object and some keep going until they find the point or region.

7.3.2. Upward Chaining

Messages start at some subview and then chain upward, stopping at any point. These are the `DoCommands`. It is very important to know

that the path the DoCommand travels is based on a supervisor relationship. It follows the rule of looking for its enclosure, the object that is in charge of it.

7.3.3. Downward Chaining

A message starts at some object and then spreads downward to all the subviews inside that object. The message can start anywhere in the system. Examples include DoDraw messages and DoSetEnvironment.

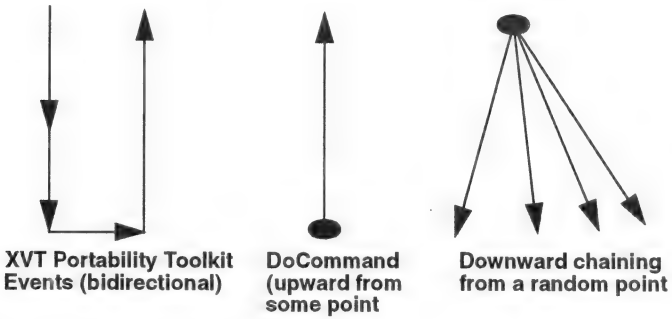


Figure 13.Channels of Message Passing

7.3.4. The Role of CBoss

Tying the entire XVT-Power++ application framework together is a class called CBoss. More specifically, every class in the application framework derives from CBoss, which defines the basic messages that any object in the framework is capable of receiving. First, CBoss provides access to global, shared objects and global, shared data. Next, it defines some messages that can be passed along or delegated through the levels of the application framework.

7.3.4.1. DoCommand Messages

The most often used type of message is the DoCommand. Any object in the XVT-Power++ application framework can receive a DoCommand message. The DoCommand allows you to send any generic message to any object. Two items of information can be passed along with a DoCommand message:

```
virtual void DoCommand(long theCommand, void*
theData=NULL);
```

The first is a command ID number. This parameter is of type `long` and can take any number. You are responsible for managing the ID numbers that you give different commands. XVT-Power++ reserves some predefined, internal XVT-Power++ command ID numbers. As is noted in the section on `CGlobal` in the *XVT-Power++ Reference*, the XVT-Power++ ID number base is 20,000. All user-defined ID numbers should be lower than this base. In addition to the ID number, you have the option of passing a pointer to any object you desire along with the `DoCommand`.

By default, `DoCommands` are used for click events. Views trap the logic of `MouseDown`, `MouseUp`, `MouseMove`, and `MouseDouble` events to determine whether a click has occurred inside them. If a click has occurred, a view generates a `DoCommand` message that is sent upwards through the application framework.

7.3.4.2. ChangeFont Messages

Any object can receive a `ChangeFont` event when a user selects a change of font from the menubar. This font change can be handled on any level of XVT-Power++'s application framework: at the view level as a particular view changes its font, at the document level as all of the views associated with a document change their font, at the application level as every view of every document in the application changes its font.

When a user selects a change of font, a `ChangeFont` message is sent to the window from which the menu selection was made. The window either sends the message to the selected view, if there is one; changes its own font; or propagates the message up to its document. The document may either change its own font or propagate the message up to the application. By default, the messages are propagated upwards until a non-inherited environment is encountered. For example, if a document is not inheriting an environment, it changes its own font. You can, of course, override `ChangeFont` to define a different logic.

7.3.4.3. DoMenuCommand Messages

Similarly, `DoMenuCommand` messages can be propagated and delegated from one object to another, from a window to its document and on up to the application if no view object handles it. This type of message is sent when a user selects a generic item from the menubar, perhaps a user-defined item.

7.3.4.4. Unit Messages

You can specify the units of measure that are used by any object in your application— such as inches, centimeters, characters, or a user-defined unit—as an alternative to the XVT-Power++ default pixel units. You can propagate "update unit" messages to notify objects that the units have changed and that they must recalculate measurements based on the units.

7.4. Setting Up Menus and Handling Menu Commands

We have already noted that `DoMenuCommand` is first called for the window from which the menu item is selected. The `DoMenuCommand` method has parameters for specifying which menu item was selected and whether the Shift or Control key was pressed as well:

```
virtual void DoMenuCommand(MENU_TAG theMenuItem,
                           BOOLEAN isShiftKey,
                           BOOLEAN isControlKey);
```

`theMenuItem` is the menu item number of the command, starting at one. Disabled commands and dividing lines count in the numbering.

Setting up menus is another task that can be done at different levels in the XVT-Power++ application framework. Obviously, you can set the menus of a particular window.

See Also: Chapter 8 contains a detailed discussion of what it means to have a menubar on a window.

When you want the menubars to be consistent for an entire document, the `CDocument` object should be in charge of setting up the menubars for all the windows associated with it. When you want the menubars of all the windows in an application to be consistent, the `CApplication` object should be in charge of setting up the menubars.

`CApplication`, `CDocument`, and `CView` all have methods called `SetUpMenus` and `UpdateMenus`. It is important to understand the difference between setting up menus and updating menus on the different levels. Setting up menus is a task that is done once—when an application is executed and a new window is brought up. For every window that is created, `SetUpMenus` is called. On the other hand, `UpdateMenus` is called every time a window comes to the front of the window stack and its menubar needs to be updated.

When a window is created, SetUpMenus is called for that CWindow object, for the CDocument object that instantiated the window, and for the CApplication object. Each level gets a chance to set up the window's menubar. When the window comes to the front, the menubar can be updated, although there is usually nothing to be done because the state of a window's menubar is saved when the window is relegated to the background. When the window is brought to the front again, its menubar in its saved state is put back up. Sometimes, however, you may want to update it because changes have occurred while the window was in the background.

7.5. Handling Keyboard Events

By default, keyboard events are handled the same way as menu events. When a keyboard events comes in to a window, it either goes to a view that is set as the keyboard focus for that window or it is propagated on up to the document and perhaps to the application—whichever level can take care of it. A keyboard event goes directly to the application if no windows are open.

CSubview has a method called SetKeyFocus for setting the view that is to receive the keyboard event. This view then receives the keyboard input, regardless of which view is at the front. Setting the key focus is necessary because, unlike mouse input which uses the mouse cursor to point to a specific screen coordinate, keyboard input does not clearly point to the view to which it is directed.

7.6. Setting The Environment

The "look-and-feel" is another aspect of a graphical application that is determined on all three levels of XVT-Power++'s application framework, through XVT-Power++'s CEnvironment class. CEnvironment allows you to give the various windows and views in your interface a consistent look-and-feel while reserving the option to make the look of a view or set of views as distinctive as you like. Through CEnvironment, you set such display properties of objects as their foreground and background colors, font type, line width, brush pattern, drawing mode, and anything that pertains to displaying an object that can have different attributes on the screen.

By default, there is a global environment object that is shared by every displayable object in an XVT-Power++ application. An environment object can be attached at any level in XVT-Power++, all the way from the application object to the deepest subview. An environment propagates downward, so many objects can share the

same environment. If you change a property of an environment, then every object that is sharing it is affected. Each object that is using the environment for its own drawing gets an update message notifying it of the change. For example, if the font changes, a text view must resize itself to accommodate the new font.

You must create environment objects when you want specific environments for different documents. One document might have an environment with a blue background while another has a yellow one. If there is a document environment, the document's windows use the document environment as their own. While the document is not displayable, all windows and other views associated with it display themselves in its background color. If the document does not have an environment, then it uses the global application environment stored in `CGlobalClassLib` (see Figure 14).

Each `window` can have its own environment object, with its own colors and fonts. If it does, it uses the environment and passes it down. If it does not, it uses the document environment. You can attach a different environment object to each of the windows—or to the windows that you want to have specific environments. You can go still further, allowing some of the views in a window to have their own environment objects. The subviews of those views, too, can have their own environment objects (see Figure 14).

Typically, you do not want to give environment objects to each view because many of the views can share the same environment, eliminating a lot of storage and overhead. Attaching an environment at a higher point allows every object to share it from there on down.

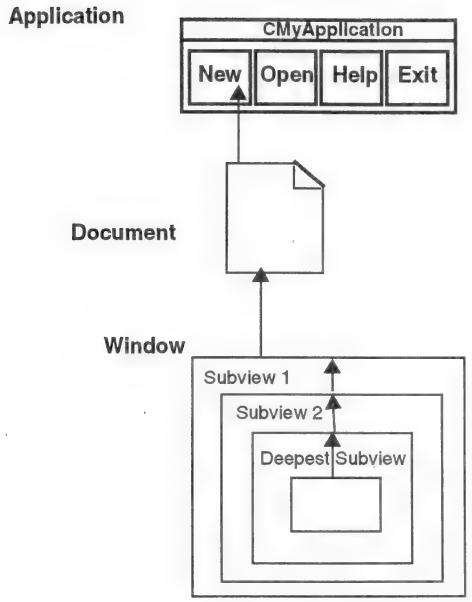


Figure 14. The Use of Environment Objects in the XVT-Power++ Object Hierarchy

7.7. Setting The Units Of Measure

As described in Section 6.5.4, XVT-Power++'s `CUnits` class enables you to set the size of the units of measure used in your application, with the options being pixels (the default), inches, centimeters, characters, or a user-defined unit.

Units of measure are treated in exactly the same way as environment properties. That is, by default there is a global application `CUnits` object that propagates through all levels of the XVT-Power++ application framework to the deepest subview. However, you can set the units of measure at different levels: for different documents, windows, and view enclosures. When you change the units of a view, all objects sharing its `CUnits` object are affected and must accommodate the change.

7.8. Printing

At any point, you can notify any object in XVT-Power++'s application framework to print and it will place itself into the print queue and print out on the designated printer. Upon receiving a print command

- A window prints all views inside of it
- A document sends every window associated with it to the printer, which prints each window on a separate page
- The application notifies all of its documents to print all of their windows, each on a separate page

By default, whatever is drawn on the screen is drawn on the printer paper. However, you may want a given object—say, a window—that draws a certain way on the screen to draw differently on the printer, perhaps using a different font. In this case, you can override the `CView PrintDraw` method for a view to define how you want it to print. `PrintDraw` is a method that is in charge of doing any drawing that should be sent to the printer. This drawing is just like the drawing that is done on the screen using the XVT Portability Toolkit `win_draw` functions. If you were to write your own printer `Draw` method, it would look just as if you were printing from the screen. You would still print using the view's XVT Portability Toolkit window, which can be obtained through the `CWindow GetXVTWindow` method. `GetXVTWindow` can return either a regular screen window or a printed window, depending on whether printing is being done. `PrintDraw` just calls the regular `Draw` method, and in most cases this is adequate.

7.9. The Shell Classes and the Shell Utility

As we noted at the beginning of this chapter, the advantage of XVT-Power++'s application framework is that it enables the application developer to reuse the basic structure and design of a program. Nothing illustrates this capability more than XVT-Power++'s shell application classes: `CShellApp` (a shell application), `CShellDoc` (a shell document), and `CShellWin` (a shell window). When you compile these classes together with the application framework, they produce a simple running application with an open window. The entire event handling and message passing structure and the procedures for execution are in place. XVT-Power++ has already done the work to provide you with this functionality, freeing you to concentrate on adding your own specific logic, defining the look of

your application's windows and views, defining the types of data that the application will manage, and so on.

XVT-Power++ supplies basic utilities that automatically create a shell application for you. These utilities are located in the **demo/shell** directory. In most cases you should use these files when starting a new application rather than starting from scratch. For information about the utility for your platform, see the Installation section.

8

WINDOWS

8.1. Introduction

Windows are a special type of view in XVT-Power++ because although they inherit from `CSubview`, they have a predefined enclosure. This enclosure is either the screen itself or, on platforms such as Windows or Presentation Manager, the task window. Objects of type CWindow are the only views derived from `CSubview` that cannot be nested inside other views. In fact, windows are the topmost enclosure for any other type of XVT-Power++ view, and their layout on the screen is managed by a class named `CDesktop`.

This chapter discusses the attributes a window can have, the possible types of windows, how they are constructed, and how you can derive your own classes from `CWindow`. It briefly considers two example classes of windows that are derived from `CWindow`: the shell window and the task window. The chapter concludes with a look at `CDesktop`.

8.2. Window Attributes

When you instantiate an object of type `CWindow`, you can set several different attributes that are available through the XVT Portability

Toolkit in order to tailor the window to the very specific needs of your application: deciding whether it has scrollbars or a menubar of its own and whether it can be moved, sized, or iconized. The possible window attributes are listed in the following table. XVT supplies other, platform-specific flags.:

WSF_NONE	no flags set
WSF_SIZE	is user-sizeable
WSF_CLOSE	is user-closeable
WSF_HSCROLL	has horizontal scrollbar outside client area
WSF_VSCROLL	has vertical scrollbar outside client area
WSF_DECORATED	all of above four flags are set
WSF_INVISIBLE	is initially invisible
WSF_DISABLED	is initially disabled
WSF_ICONIZABLE	is iconizable
WSF_ICONIZED	is initially iconized
WSF_NO_MENUBAR	has no menubar of its own
WSF_MAXIMIZED	initially maximized

If the window has a scrollbar, you can choose between horizontal or vertical scrolling; if it has a menubar, you must decide how it will react to the menubar's events when the user selects it. Of course, you will want to ensure that the window can respond to events that come through the system—such as mouse clicks, drawing events, moving to the front of the window stack, and so on—unless you purposely disable it.

8.3. Interaction With the Document

A `CWindow` object can be of any XVT type; it receives all window events. Most of the window management, such as moving and sizing, is done by the window manager or the XVT-Power++ desktop. Possible XVT window types are shown in the following table. For further information, see the *XVT Programmer's Guide*.

W_DOC	document window
W_PLAIN	single-bordered window
W_DBL	double-bordered window
W_NONE	no std_win ever

W_NO_BORDER	no border

Windows are normally created inside `CDocument`'s `BuildWindow` method and are in charge of viewing the data that the document is managing. Each window has access to its own document object through a pointer, and interacts with it by default at several different points in the life of the application. For example, when the window is going to close, it checks to see whether the data in the document needs to be saved. If it does, the window invokes a dialog box giving the user the option to either to save the data, close without saving, or cancel the close operation. Windows also interact with their documents when they delegate tasks to them. For example, if a window receives a `DoCommand` and elects not to trap it, it delegates the `DoCommand` up the chain of the application framework to its document. This is similarly true of the key events and menu events described in Chapter 7.

8.4. Window Construction

▼ Normally, window construction occurs as follows:

1. You derive your own application-specific window class.
2. You add some objects that will be nested inside the window, creating them in the window's constructor.
3. You give the window itself as the enclosure of the objects that will be nested inside.

For example, if you want to create a window that has three buttons and a text field object, you would instantiate three buttons and a text field in the `CWindow` constructor and give the constructor the `this` pointer as the enclosure of those objects, as follows:

```
MyWindowClass::MyWindowClass(CDocument* theDocument,
const CRect& theRegion)

    : CWindow(theDocument, theRegion, "", NULL, W_DOC,
NULL)
{
    NButton* aButton = new NButton(this,
CRect(10,10,60,50), "Button1");
    ...
    aButton = new NButton(this, CRect(60,10,100,60),
"Button2");
    ...
    aButton = new NButton(this, CRect(110,10,150,60),
"Button3");
    ...
    NLineText* aField = new NLineText(this,
CPoint(10,80), 140);
}
```

Once this specific class is defined and you want to create a window with three buttons and a text field, you simply instantiate your new window class, which is derived from `CWindow`.

To destroy a window, call the `Close` method on it. In response, the window disappears and the memory that it occupies is freed. Closing a window is semantically equivalent to deleting the window. If you have a pointer to a window object, `window object -> close` is equivalent to `delete window object`, except that you are not allowed to say "delete window object" because the constructor of `CWindow` is protected.

8.5. The Shell Window

The shell window (`CShellWin`) is an example class that is part of the shell application framework discussed in Section 7.9; it overrides a minimum number of aspects of `CWindow` for demonstration purposes. `CShellWin` does not add any more objects to the window, but simply shows an empty derived class that you can use as a starting point.

8.6. The Task Window

The task window, `CTaskWin`, is a class that XVT-Power++ uses internally. It is a private class that can be instantiated only by XVT-Power++. This class is created for use on platforms that require a task window to enclose all other windows in the application. `CTaskWin` maps directly to the XVT task window representation, `TASK_WIN`.

A task window has several of the properties of other windows, but it is much more limited in what it can do, especially since it is confined to certain platforms. Even within those platforms, the limitations on the task window depend upon whether the window is drawable, a property that can be toggled through XVT's `set_value` function.

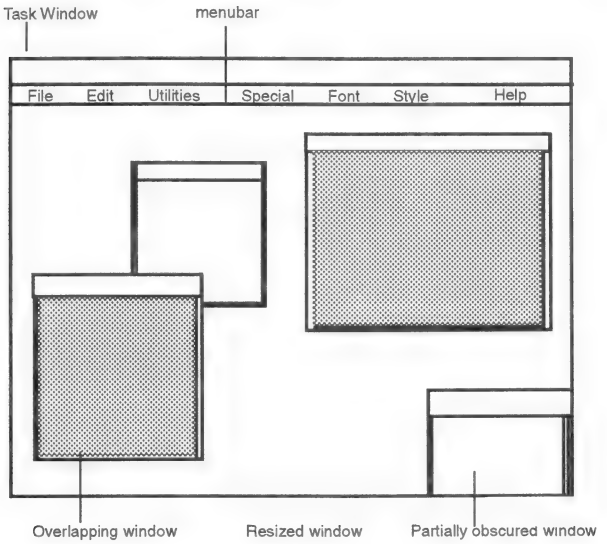


Figure 15. Example Task Window

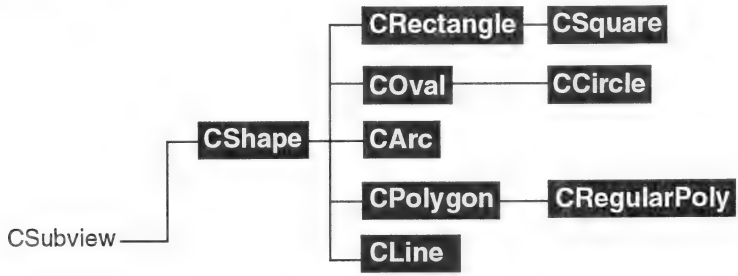
8.7. The Desktop

No discussion of `CWindow` is complete without a consideration of `CDesktop`, which is responsible for managing the layout of the different windows on the screen. At any time while it is running, an XVT-Power++ application will have a number of windows open on the screen, without regard to their associated documents. The desktop keeps track of all these windows and their active/inactive states. The desktop is notified each time a different window is brought to the front of the window stack. It has methods for setting and getting the front window and a method for placing a window on the screen, staggering it with windows that are already present. Also, you can use `CDesktop::FindWindow` to find out whether a certain window is in the desktop and `GetNumWindows` to find out how many windows are currently in the desktop. Finally, `CDesktop` has protected methods for adding and removing a window from the desktop. These methods can be accessed only by `CDocument` objects.

9

SHAPES

9.1. Introduction



XVT-Power++ includes a wide range of shape classes that allow you to draw different shapes on the screen. Since all of the shape classes derive from `CShape`, which in turn derives from `CSubview`, they have the properties of subviews. Each shape is an object that can act as an enclosure for other views, receive events, be moved or sized, have its own clipping area, contain its own coordinate system, and so on. You can use shapes not only as decorations for windows but also as buttons or other types of objects that generate commands and allow the user to interact with the application.

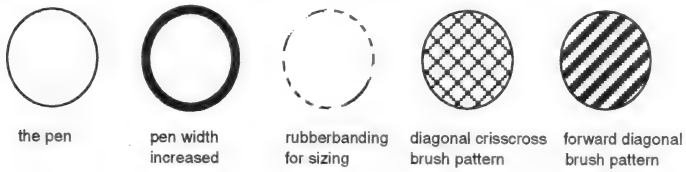
This chapter surveys the different kinds of shape objects available in XVT-Power++, considers the resources for drawing them, and offers guidelines for when you can most appropriately use them rather than directly calling the XVT Portability Toolkit's drawing functions.

9.2. Use of CEnvironment for Drawing

The shape classes use `CEnvironment` for drawing purposes. The border of a shape is drawn with the pen, and its interior is painted

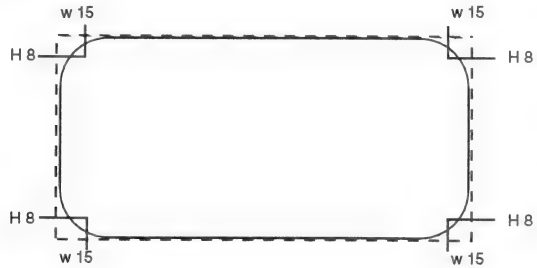
with the brush. You can set the color and pattern of both the pen and the brush. Also, you can set the pen width.

See Also: For details on the available colors, brush patterns, and pen patterns, see `CEnvironment` in the *XVT-Power++ Reference*.



9.3. Rectangles and Squares

An interesting feature of the `CRectangle` class is that rectangles can optionally have rounded corners. Specified values for the width and height of the corners indicate how high and how deep the rounding should be.

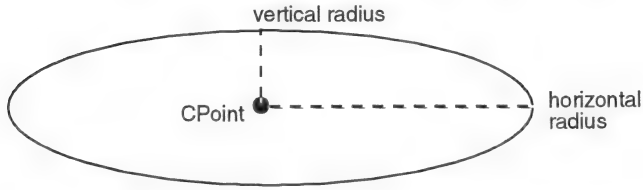


The `CSquare` class, which derives from `CRectangle`, creates a square that, like its parent, can have rounded corners. If you size a square and do not specify an equal height and width, the `CSquare` class takes the average to calculate the square's new size.

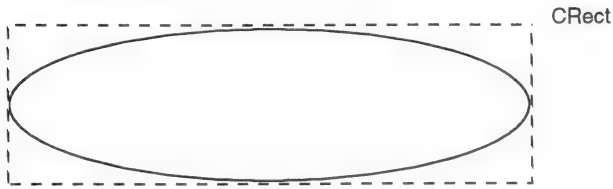
9.4. Ovals and Circles

When you instantiate an object of the `COval` class to create an oval shape, you can construct it in one of two ways:

- give it a center point and a horizontal and vertical radius



- specify a region (`CRect`) within its enclosure that is used to place the oval



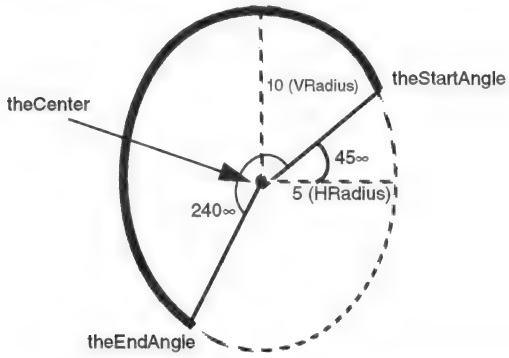
Unlike the `COval` class from which it derives, `CCircle` requires only one radius because all points are equidistant from the center.

9.5. Arcs

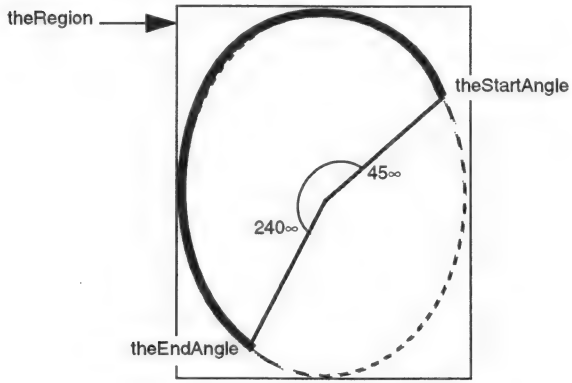
`CArc` is analogous to the `COval` class, except that you specify a starting and ending angle for drawing the arc. You can also give the arc an interior fill so that you are drawing a piece of pie rather than an arc.

Like the oval, an arc can be constructed in one of two ways; in fact, the arc is drawn counterclockwise along an implicit oval from one given angle to another:

- give it a center point, horizontal and vertical radii, and starting/ ending angles



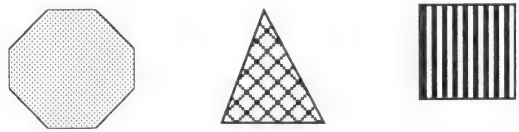
- specify a region (`CRect`) within its enclosure that is used to place the arc



9.6. Polygons

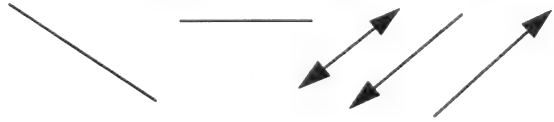
`CPolygon` is a class that would be more aptly named "`CPolyline`" because it does not necessarily create a polygon. Basically, you give this class a set of points, and these points are connected.

Deriving from `CPolygon` is `CRegularPoly`, which draws a true polygon, calculating the positions where the points should be connected in order to construct a regular polygon—a triangle, a square, a pentagon, and so on. You provide the number of sides and a certain radius, and perhaps a rotation angle, and `CRegularPoly` draws the shape for you.



9.7. Lines

XVT-Power++'s `CLine` class draws a line inside a view enclosure. Like the other shape classes, `CLine` brings with it all the freight of a `CSubview`. A line can receive events, generate commands, have different types of stickiness properties (see `CGlue`), and so on. Lines can also have beginning and/or ending arrows.



9.8. Drawing Shapes in XVT-Power++

XVT-Power++ users may be unsure about when to use the different shape classes to draw objects on the screen versus overriding the

`Draw` method of a view and using the XVT Portability Toolkit drawing functions to draw lines, squares, ovals, and so on. As noted earlier, XVT-Power++'s shape classes enable you to draw shape `objects` derived from `CSubview` with all the properties of `CSubview`. Obviously, there is more overhead to using one of the shape classes than there is if you just draw a shape directly by calling one of XVT Portability Toolkit's drawing functions, such as the function for drawing a line. In short, you need a rule of thumb to determine when it is appropriate to draw from XVT-Power++ rather than the XVT Portability Toolkit, and vice versa.

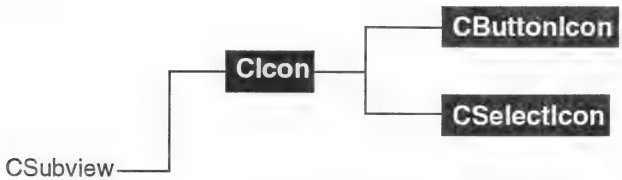
If you are creating a complex view that is derived from `CView` or `CSubview` and that contains several kinds of intricate drawings, many lines, and so on, we recommend that you use direct the XVT Portability Toolkit function calls. The drawing process will be much faster and involve much less overhead. If, on the other hand, you want your drawings to have certain behaviors, such as moving/sizing capabilities or the ability to generate commands like a button or nest other objects, you can readily get access to these behaviors using inherited XVT-Power++ code. These are design decisions that you must make while developing your XVT-Power++ application.

Another design issue is the seeming awkwardness of some XVT-Power++ shapes when functioning as enclosures. For example, you may wonder what could fit inside a line. Actually, the enclosure region of a line includes an imaginary box around the line from one end of it to the other. Thus, the region of the line can easily contain a label or similar object. This is also true of `CPolygon` objects, where the smallest imaginary box that could enclose the entire object acts as the region defined for that enclosure.

10

ICONS

10.1. Introduction



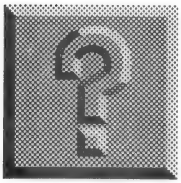
An icon resource is a bitmap picture that can be drawn on different platforms. Each platform imposes its own restrictions on how the resources are handled. For example, on some platforms, icons can appear in only two colors, while on other platforms they can be drawn in a wide range of colors. Some platforms limit the size of icons to 32 x 32 pixels, while others impose no limitations on size.

See Also: For information about the limitations imposed by the your platform, consult your *XVT Programmer's Guide*.

When you want the most portability from platform to platform, you can safely assume that icons must be drawn in only two colors and are limited in size to 32 x 32 pixels. If you are not interested in operating by the lowest common denominator, you can choose to work with each platform to its fullest extent when you draw your own resources. Keep in mind that there are several programs on the market and in the public domain that allow you to convert icon resources from one platform type to another.

This chapter describes the basic functionality of XVT-Power++ icons and then briefly looks at how two variant classes extend this functionality.

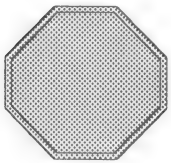
10.2. Icons in General



XVT-Power++contains a basic (but not abstract) icon class called `CIcon` that provides general icon properties for its two child classes, `CButtonIcon` and `CSelectIcon`, to inherit. `CIcon` allows you to draw an icon in either an enabled or disabled state. An enabled icon can receive events, generate `DoCommands`, and perform the other functions of an XVT-Power++ view object. If an icon is disabled, it cannot receive events. `CIcon` derives from `CSubview`, so icons have all the properties of subviews, including the ability to act as an enclosure for other objects. Thus, you can nest a text object or a miniature picture inside an icon if you wish. All types of icons can have a title, which is centered beneath the icon. In fact, the entire enclosure area of an icon encompasses the smallest rectangle that covers both the icon and its title.

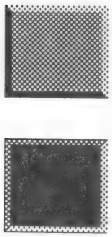
When you create an icon, it takes two resource IDs for bitmaps showing the icon in its enabled and disabled states. If you do not need both states, you can pass in the same resource ID twice. `CIcon`'s child classes, `CButtonIcon` and `CSelectIcon`, extend the basic icon behavior in a couple of useful ways.

10.3. Button Icons



XVT-Power++'s `CButtonIcon` class adds button behavior to an icon. When a user clicks on a button icon, it produces a command and generates a response, and it also becomes indented or inverted. Thus, it requires three resource IDs: one for its enabled state, one for its "pressed" state, and one for its disabled state.

10.4. Select Icons



XVT-Power++ `CSelectIcon` class provides an icon that acts as a selection box or a check box. When a user clicks on an icon of this type, it becomes selected and remains in its selected state until the user clicks on it again. That is, it generates a select command when clicked and a deselect command when clicked again. A `CSelectIcon` object takes three resource IDs: one for its enabled state, one for its disabled state, and one for its checked or selected state.

10.5. Environment Settings for Icons

Under some platforms, the icon drawing appears in the foreground color, as does its title. Open spaces in the drawing appear in the background color. You can set the background and foreground colors. Under other platforms, the icon's color is fixed as defined. For portability, set the colors appropriately even if the information is not used.

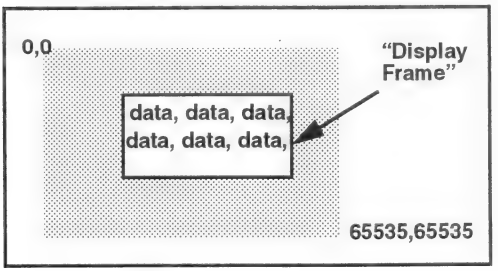
11

VIRTUAL FRAMES

11.1. Introduction

CSubview — **CVirtualFrame** — **CScroller** — **CListBox**

When you gaze out of an office window, you can see only a portion of the scenery surrounding the building. As you move your head about, different trees, buildings, or areas of the parkinglot become visible through the window frame, depending on the angle of your view. Similarly, when you are programming a graphical user interface, what you want to display is often too large to fit into the display area on the screen. The object to be displayed might be a window or just a portion of view inside a window. To allow you to bring different parts of a large object into view, XVT-Power++ provides virtual frames. A *virtual frame* consists of a large virtual area into which you can insert various types of view objects and a smaller display frame through which only a certain portion of the area is visible at a given time.



11.2. The CVirtualframe Class

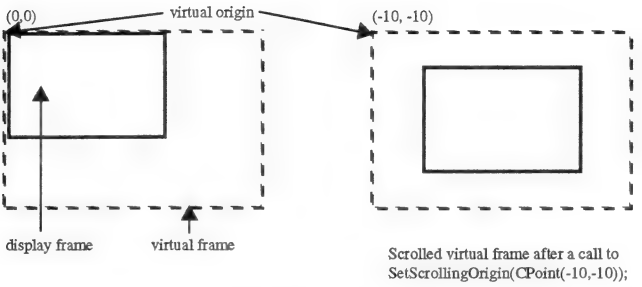
XVT-Power++'s `CVirtualFrame` class is an abstract class that provides mechanisms for viewing different areas of a virtual frame. For example, you can call the `ScrollViews` method, which scrolls the virtual area to the right, bottom, top, and so on. However, `CVirtualFrame` has no mechanism that allows the user to scroll the virtual frame directly. This mechanism must be provided by such derived classes as `CScroller`, which attaches scrollbars to the display area of the virtual frame. A user manipulates the underlying virtual area by means of these scrollbars to bring different parts of it into view. That is, `CScroller` automatically calls `ScrollViews` whenever necessary. You can write other classes that offer a different kind of scrolling mechanism, perhaps a virtual pane that a user can drag to move the contents of the virtual area. You may prefer to create navigation buttons that a user can press to move around inside the virtual area.

11.2.1. Automatic Sizing Capabilities

When you create a `CVirtualFrame`, you can set the size of the virtual area as well as the size of the visible area. If you do not set the size of the virtual area, then it initially has the same size as the display area. As a user inserts objects into the virtual area, it expands, if necessary, to ensure that all objects nested within it can fit inside the virtual area even if they do not fit inside the display area. `CVirtualFrame` contains methods such as `EnlargeToFit` and `ShrinkToFit` that automatically size the virtual area to fit a certain number of enclosed views as objects are added or removed.

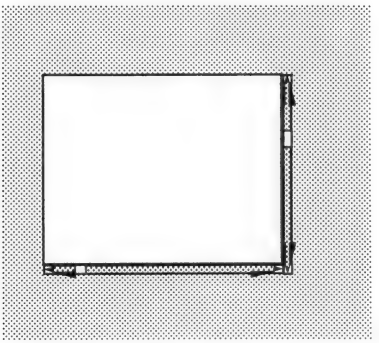
11.2.2. The Scroll Range

Tied to the automatic sizing capabilities of the virtual frame is the idea of both a vertical and a horizontal scroll range. A `scroll range` specifies the range that is allowed for scrolling a given virtual frame. If a virtual frame has a virtual area that is twice the size of its display area, then its range might start at the top of the display area and end at the bottom of the virtual area to define the entire range of scrolling that is possible. Included with this range is a maximum and minimum position as well as an origin position indicating where the display area is currently located relative to the virtual area.



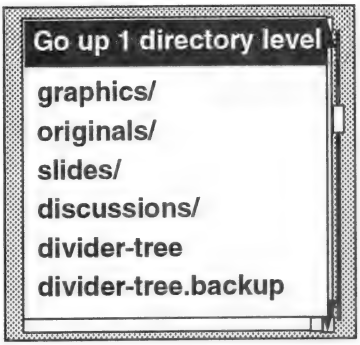
As the virtual frame sizes itself relative to the display area, the scrolling range varies. Thus, `CVirtualFrame` has two pure virtual methods, called `SetHScrollRange` and `SetVScrollRange`, that are called whenever the range changes. These methods must be overridden by derived classes. For example, `CScroller` overrides these two methods to set a scrollbar's position or thumb proportion depending on the range of scrolling that is possible inside the virtual frame.

11.3. The CScroller Class



The `CScroller` class derives from `CVirtualFrame` and adds scrollbars to the display area of a virtual frame. It is a virtual frame with scrollbars. These scrollbars allow the user to manipulate the virtual area. A `CScroller` object can have a horizontal and/or a vertical scrollbar. When users drag or size objects inside a scroller, the contents scroll automatically as the object is dragged beyond the visible borders.

11.4. The CListbox Class



The `CListbox` class derives from `CScroller`, providing a scrollable box that contains a list of selectable text items. A `CListBox` object is a composite of the `CScroller` and `CGrid` classes. The scroller contains a grid into which the text items are inserted. Of course, `CListBox` has several utility methods for inserting, removing, selecting, and deselecting the text items in its grid. If you have a large amount of text, we recommend that you insert this text when you initialize the list box, giving it a list of `CString` objects. Giving a list box a list of text upon initialization is better than going through a loop calling `InsertLine` because every time a new line is inserted, the thumb positions are adjusted if necessary. Thus, if you insert a hundred items, the thumb will be adjusted a hundred times and you may notice some flashing on the screen.

At any time, you can use `GetSelectedLine` to find out which line in the list box, if any, is selected. This method returns either the number of the line that is selected or a minus one (`-1`) if no line is selected. In addition to XVT-Power++'s list box, there is a native list box that is constructed through the native toolkit. The advantage of using `CListBox` is that you can modify and extend it, perhaps arranging the text items into two or three columns or deriving a list box class that displays picture items instead of `CStrings`.

11.5. Use of the Environment

The borders of both `CScroller` and `CListBox` are drawn with the pen, and the interiors are painted with the brush. You can set the color, pattern, and width of the pen. Also, you can set the brush color for both `CScroller` and `CListBox`. Finally, you can set the

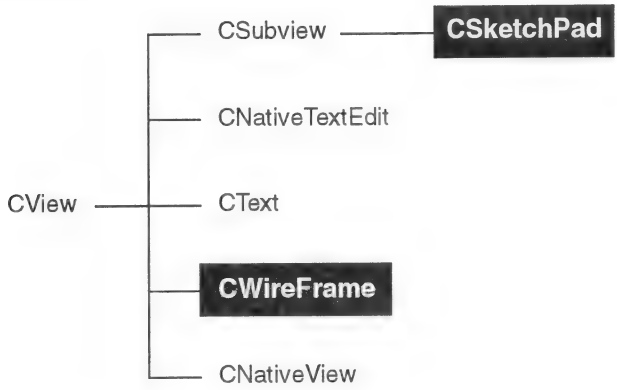
foreground and background colors. In addition, keep the following points in mind:

- The text items contained in the list box are drawn in the background color, but, when selected, are drawn in the foreground color.
- The environment properties of the scrollbars are system-defined.
- For `CScroller`, the brush pattern has been disabled and is always `PAT_SOLID`, even if you give it another setting. However, you can define the brush for `CListBox`.

12

WIRE FRAMES AND SKETCHPADS

12.1. Introduction



One of the features of XVT-Power++'s view classes is that you can easily make them movable and/or sizable. When you click on a view to select it, a rubberband frame surrounds the view, enabling you to drag the mouse to change the view's size or move it to another screen location. Moving and sizing are handled automatically, and the mechanisms that make these operations possible are all embedded into one class: `CWireFrame`. Related to `CWireFrame` is `CSketchPad`, which uses the wire frame to sketch shapes within a drawing area on the screen. When a user drags the mouse across a sketchpad's drawing area, a rectangular wire frame appears and stretches with the mouse. When the user releases the mouse button after creating, sizing, or moving a drawing on the sketchpad, the wire frame disappears. The wire frame can also act as a selection box, allowing you to drag out a rubberband frame that selects every

object inside it. This chapter discusses useful features of the CWireFrame and CSketchPad classes.

12.2. Wire Frames

The CWireFrame class acts as a friend class. When a view is set to be sizeable or draggable, it instantiates a helping CWireFrame object to enable the appropriate behavior. If an object is movable or sizeable and thus has a helper wire frame, this object is in effect disabled. The object will no longer receive any mouse events. Instead, the mouse events are sent to the wire frame that it owns. For example, a button that normally invokes a dialog box when it is pressed will not do so if it is made moveable. Instead, its wire frame receives the event so that the button can be moved to another location on its enclosing window.

XVT-Power++ takes care of most of the functionality of CWireFrame internally, and you will rarely have to deal with this class directly. However, when developing your application, you may decide to change the behavior or the appearance of a wire frame by deriving a new class from CWireFrame. After instantiating the new class, you would use CView's SetWireFrame method to set a given view's wire frame to the new one instead of the default one.

XVT-Power++ offers more than one wire frame class. CWireFrame has two child classes, CHWireFrame and CVWireFrame, that allow the user to drag the mouse only in a horizontal or a vertical direction and which serve as examples of how you can override CWireFrame. For more information on these classes, see their respective sections in the *XVT-Power++ Reference*.

12.2.1. Selection and Multiple Selection

XVT-Power++ allows you to select several views and move them around on the screen simultaneously. Clicking on a view to select it causes any previously selected view(s) to become deselected. However, if you press and hold down the Shift key while clicking on a view that own a wire frame, you can select multiple views simultaneously. A group of nested views moves together inside the top-level enclosure, regardless of which of them you may have selected.

12.2.2. DoCommands

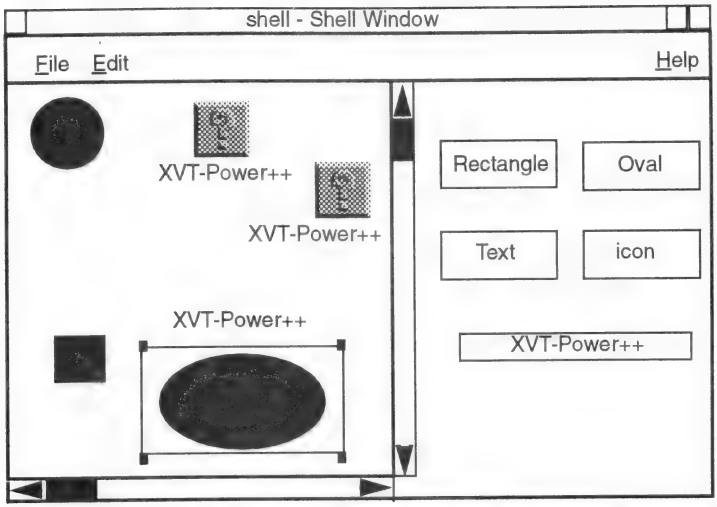
A CWireFrame object generates internal XVT-Power++ commands through the DoCommand mechanism in response to certain events. For example, when a view is selected, a CWireFrame Select command is generated, and when the view is deselected, a CWireFrame Deselect command is generated. When a view is sized a CWireFrame Size event is generated. Along with the command, the DoCommand takes a pointer to the object that was moved, selected, sized, or so on.

12.2.3. Drawing

Two kinds of drawing occur for a wire frame. Thus, if you want to change the look-and-feel of the wire frame, you must override two CWireFrame drawing methods. First is the drawing of the wire frame itself, which occurs inside DrawWireFrame. Second, CWireFrame draws the wire frame's handles inside DrawFrameGrabbers. These are the handles that appear on the wire frame when it is selected. Dragging these handles with the mouse, you can resize the wire frame's owner.

12.3. Sketchpads

XVT-Power++'s CSketchPad class works in conjunction with CWireFrame to provide the drawing functionality that users typically expect from a graphical user interface. You can use it to draw objects on the fly and to select objects that have been drawn. On the drawing area, you can drag out either a rectangular wire frame or a line wire frame between the MouseDown point and the MouseUp point. The line sketching is done through a class derived from CWireFrame that draws lines rather than rectangles.



XVT-Power++ objects, with **COval** selected. Each time the user clicks on one of the button, a new object appears inside the scroller. The objects can be selected by a mouse click. Once selected, objects can be dragged and sized. Objects are deselected when the user either clicks on the background area or selects another object.

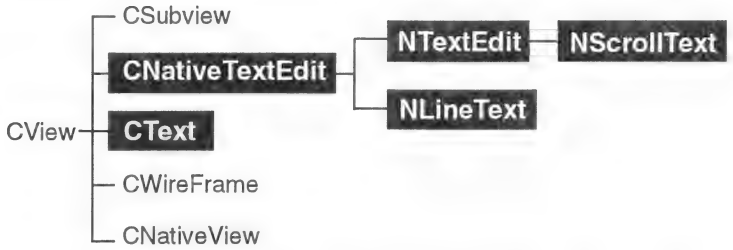
Figure 16. Window With a Sketchpad Embedded Inside a Scroller

When you release the mouse button and the wire frame disappears, a `Docommand` is generated. Your application should react to the command by calling one of `CSketchPad`'s sketch event methods, which are described in detail in the *XVT-Power++ Reference*. For example, you can call `GetSketchedRegion` to find out the coordinates of the region that was sketched and then use this information as appropriate for your application. You can also use `SetSketchEverywhere` to specify whether the sketchpad itself will receive the events or whether the objects drawn in it will receive the events. This is an important point because it determines the basic behavior of the sketchpad. If the sketchpad receives the events, the user can draw overlapping objects and even sketch one object directly on top of another. However, if the objects themselves are receiving the events, drawing can occur only over the empty space within the sketchpad.

13

TEXT AND TEXT EDITING

13.1. Introduction



XVT-Power++ provides two overall text facilities. One is `CText`, XVT-Power++'s static text drawing class. The other is a set of native text editing classes that harness the text editing capabilities of the XVT Portability Toolkit. These classes are "native" classes because the implementation of the actual text editing is done by the XVT Portability Toolkit. XVT-Power++ provides some extra features and encapsulates a lot of the work involved in using the text editing objects, but the native objects are implemented by the XVT Portability Toolkit rather than by XVT-Power++. Both `CText` and the native text editing facilities allow you to choose from a variety of font families (Courier, Helvetica, Times, and so on), styles (italics, bold, and so on) and sizes (in points). `CText` has a `SetFont` method through which you can set the font of a `CText` object. The platform on which you are working determines the availability of different fonts. You can also set a font through the `CEnvironment` class, through the XVT Portability Toolkit `select_font` function, or through a font menu event.

13.2. CText

CText displays a string of read-only text that is useful for one-line instructions, button names, titles, and so on. When you instantiate a CText object, you give it a CString object, which may or may not be initialized using a string resource ID. Detailed information on using CString resources is provided in the section on CString in the *XVT-Power++ Reference*. When you give a CText object a string, it automatically sizes itself at a designated point on the screen indicating the coordinate at which the line of text starts.

Window displaying a textual view

```
itsMessage = new CText(this, CPoint(100,100), "Hello World");
```

`new CText`	creates a new CText object. The signature of the CText constructor is as follows:
`CText(CSubview* theEnclosure, const CPoint& theTopLeft, const CString& theText - NULLString);`	
`this`	specifies the enclosure of the CText object.
`CPoint(100,100)`	This is the theTopLeft parameter.It is a coordinate, relative to the CText's enclosure, where the text will be displayed.

This coordinate is local to the designated enclosure of the CText object.

When you select a CText object by sending it a Select message, it inverts its colors. For example, XVT-Power++'s CListBox class provides an object that consists of a set of CText objects displayed as a list inside a grid. When the user clicks on one of the items, it receives a Select message and becomes highlighted.

13.3. The Native Text Editing Classes

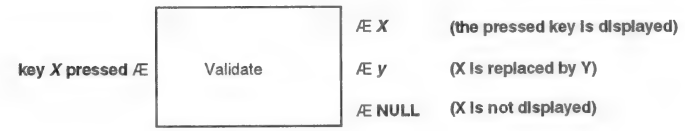
All of the native text editing classes derive from an abstract class called `CNativeTextEdit`, which has methods for setting/getting, selecting/deselecting, cutting, copying, and pasting text, and many other editing operations. As an abstract class, `CNativeTextEdit` supplies no means to organize text into lines and paragraphs or to scroll text. These concepts are embodied in its three child classes, discussed in the following section.

13.3.1. NLineText, NTextEdit, and NScrollText

`NLineText` is the simplest of the native text editing classes. It allows you to create a one-line text editing field. You give it the length of the line, and the height is calculated in terms of the font's size. `NTextEdit` is organized into paragraphs, lines, and characters in a line. This class has methods for setting/getting paragraphs or lines, and so on. When the font of an `NLineText` object (or a `CText` object) changes, the text box changes size to accommodate the new font. However, if the font size of an `NTextEdit` object changes, the object does not change size but the text inside of it changes so that more or less of it becomes visible. Finally, `NScrollText` is a class that is derived from `NTextEdit` and thus includes all of its paragraph organization features while adding the further feature of scrollbars. All of the scrolling is done automatically, and the scrollbars are updated.

13.3.2. Text Validation

Whenever a text box receives a keyboard event, a `CNativeTextEdit` `Validate` method is called, which determines how each character is to be displayed. `Validate` can opt to display the character, map the character to some other character, or not display it and return `NULL`, as shown in the following diagram.

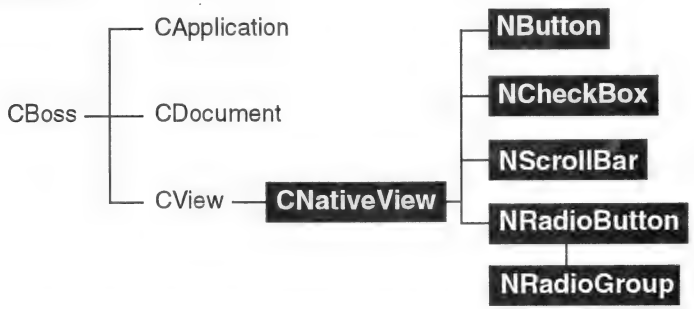


Note that there is a detailed discussion of `Validate` in the section on `CNativeTextEdit` in the *XVT-Power++ Reference*.

14

NATIVE VIEWS

14.1. Introduction



Native views are views that have the look-and-feel of graphical objects provided by the native window manager. "Native views" is XVT-Power++'s term for controls, such as scrollbars, buttons, list boxes, radio buttons, check boxes, and pop-down menus, all of which provide some means for the user to interact with the application. They are standard items on almost any GUI but look a little different from platform to platform. XVT-Power++'s native views "fit in" visually and functionally with the analogous graphical items on your platform, whether you are working in Motif or OPEN LOOK, in Windows, or on a Macintosh machine. They are thus not implemented by XVT-Power++ but, at the lower level, are implemented by the native toolkits. Thus, there is less flexibility in what you can do with them. For example, the native classes are derived directly from `CView` and thus cannot act as enclosures for other views. You cannot draw anything inside them and would not expect to on most platforms.

However, when you instantiate one of XVT-Power++'s native view classes, you do get a lot of extra features that you might not expect

to get out of controls. Since native views derive from the CView class, they have all of the capabilities of other objects at the view level. That is, they automatically propagate events, and they can be enabled/disabled, shown/hidden, activated/deactivated, moved, and sized as discussed in section 6.3. All of XVT-Power++'s native views have a DoHit method that functions as the interface for the actual events the native views can receive. Every native view class takes care of these events automatically, and you do not have to do anything about the DoHit. However, if you want to interact with the events at the lower level and create your own native view class, you may need to override DoHit.

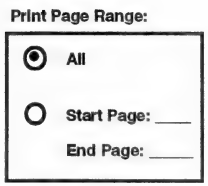
At the top of the native view hierarchy is CNativeView, which handles much of the work that must be done to manipulate the native views that inherit from it—all the way from moving and sizing to creation/destruction, enabling/disabling, and so on. CNativeView is an abstract class that cannot be instantiated because we do not know exactly which native view you want to create. XVT-Power++ provides several native view classes, and this chapter surveys them all.

See Also: For details on each class, see its respective section in the *XVT-Power++ Reference*.

14.2. Types Of Native Views

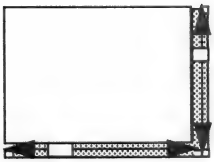
The NButton class allows you to create a button that you can "press" using the mouse, to generate a DoCommand. You must set the number of the command for the button to generate. Upon receiving a DoHit message, a button generates a command that may be handled at different levels in the XVT-Power++ application framework. When you instantiate a button, you give it a CRect to specify its size and give it a title. The title is clipped inside of the button.

The NCheckBox class provides an object that becomes selected (generating a select command) when the user clicks on it and deselected (generating a deselect command) when the user clicks on it again. You can set the ID numbers for the select and deselect commands of a check box. In addition, a check box can have a title beside it, just as a button can have a title *inside* it. You give the check box a top-left point for the title, and the entire check box is sized to fit the title. Unlike the title of a button, then, the title is not clipped to the check box. Check boxes are commonly used for menu items or items on dialog boxes so that the user can know whether an option is currently turned on or off.



Similar to the check box is the radio button, which is provided by two classes, `NRadioButton` and `CRadioGroup` because the radio button object by definition must be used in groups. While check boxes allow you to select multiple options by checking more than one box at a time, only one of the radio buttons in a group can be selected at a time. Selecting one radio button means to deselect another. The helper class `CRadioGroup` serves as a grouper. To instantiate radio buttons, you must first create a `CRadioGroup` object and then add the buttons to that group, either one-by-one or as a set. If you add them as a set, the buttons are automatically placed inside the group, vertically or horizontally, so you do not have to calculate the locations. When the radio group is instantiated, it is initially empty. You give it a point in the top-left corner to use as a reference for placing the buttons, and as radio buttons are created, the group's size increases. Each radio button is assigned a URL resource ID number. When a single button is added to a radio group, the `AddButton` method returns an integer, which is the ID number of the radio button. When multiple buttons are added to a group, the `AddButtons` method returns the ID number of the button corresponding to the first resource ID. The IDs of the remaining buttons follow sequentially.

If you indicate that you want the radio group to be drawn, it draws a box around the buttons inside it. You can have several radio button groups on the screen at one time that will act independently. When you select a radio button, it generates a `DoCommand`. As with check boxes, the titles of radio buttons are located beside the button and are not clipped—unless the radio group itself is clipped, in which case the titles will clip to that enclosure. The radio group, of course, acts as the enclosure for the radio buttons it contains, and you can nest other objects besides radio buttons within it if you desire. For example, you might want to add a text field (such as an `NLineText` object) beside a radio button or associate a small picture with it.



Another native view is XVT-Power++'s `NScrollBar`, which provides a horizontal and/or a vertical scrollbar for a view. Many `NScrollBar` objects are created automatically in XVT-Power++. For example, `CListBox`, `CScroller`, and `NScrollText` use `NScrollBar` to create scrollbars automatically. `NScrollBar` is a convenient class when you want to create your own scrollbar by instantiating one of these objects, specifying whether it is to be horizontal or vertical, and giving it a starting and ending position and a range. You can find out an `NScrollBar` object's native height and width using its `NativeWidth` and `NativeHeight` methods.

The scrollbars are updated automatically. For a vertical scrollbar, the topmost position is the *minimum* position and the bottom is the *maximum* position. For a horizontal scrollbar, the left is the minimum and the right is the maximum position. Through its `DoHit` method, an `NScrollBar` automatically captures the events it can get, such as mouse clicks or drags on its thumb or clicks for a page or line scroll. The scrollbar automatically sends these events up to its enclosure after translating them into `HScroll` and `VScroll` messages, which contain information about the type of event (line up, line down, page up, or page down) and the thumb position.

In addition to `NScrollBar`, there is a class derived from it called `NWinScrollBar`, which uses special types of scrollbars that are attached to the window. These scrollbars are special because they might have a special look-and-feel when they are attached to the window. For example, on the Macintosh if a scrollbar is attached to a window, it automatically becomes disabled when the window moves to the background. On the Windows platform, a scrollbar attached to a window disappears when there is nothing left to scroll and reappears when there is something to scroll. When you size the Program Manager, for instance, and objects are clipped, scrollbars appear automatically.

An `NWinScrollBar` object is instantiated automatically when you create a window and specify scrollbars as one of its properties. This is one of the XVT Portability Toolkit properties that you can give to a `CWindow` object. `NWinScrollBar` objects can also appear on list boxes, scrollers, and other views if they are given a special parameter that notifies them to use the window-attached scrollbar.

See Also: `NListEdit`, `NListButton`, and `NListBox` in the *XVT-Power++ Reference*.

15

GRIDS

15.1. Introduction

As a type of `CSubview`, XVT-Power++'s grid classes act as enclosures that divide a portion of the screen into rows and columns. The widths and heights of these rows and columns can be set in different ways, depending on whether the grid is fixed or variable. Each intersecting row and column of a grid forms a certain grid cell. Grids are frequently used in graphical user interfaces: in list boxes, color charts, and panels where it is important that a number of textual or graphical items be precisely placed and aligned. We encounter them in spreadsheets and drawing programs. As you design and develop your application, you will often find that grids are useful and even necessary, whether the user can see them or not. Thus, XVT-Power++'s application framework provides three classes that offer a full range of grid functionality. The base class, `CGrid`, contains a fairly extensive set of methods for manipulating grids and the objects they contain. Two variant classes allow you to create either a grid in which the cells are all the same size or a grid with variable-sized cells. This chapter gives an overview of the functionality that is available to you through these grid classes.

See Also: For details on each of these classes, consult their respective sections in the *XVT-Power++ Reference*.

15.2. Basic Grid Functionality

XVT-Power++'s abstract grid class, `CGrid`, provides methods for manipulating a grid: inserting and removing objects and placing them in different ways inside their cells, sizing the grid, getting an object from the grid either by specifying a grid location or by specifying an object, and so on. You can turn the lines and columns of a grid on or off to make the grid visible or invisible. All operations that take cell numbers or row and column numbers have a numbering system starting with zero (0) at the top-left and moving towards the bottom-right in increasing order.

15.2.1. Inserting and Removing Objects

You can insert as many views as you want into a grid cell by calling the `Insert` method. When you instantiate an object that you want to place within a grid, you must give it the grid object as its enclosure and then call the grid's `Insert` method, specifying the row and column that will contain the object. If you do not specify a row and column, `Insert` calculates a row and column location based on the coordinates of the object relative to the grid. You can also insert an object into a grid cell by calling the `Replace` method, which removes any object already present within the cell and replaces it with the new one.

If the objects nested within a grid are movable and sizable, they exhibit *snapping* behavior. As you drag an object, it snaps from cell to cell. You cannot place it between two rows or columns because it will snap into a row or column.

15.2.2. Placing an Inserted Object Within its Cell

When you insert a view into a grid cell, the view can be clipped to the cell or the view may just overlay its cell, extending beyond the cell's borders if it is too big to fit inside it. You can define how the view is placed within the cell boundaries: top-left, bottom-right, top-right, bottom-left, or justified. You can even give it an offset from each of the sides.

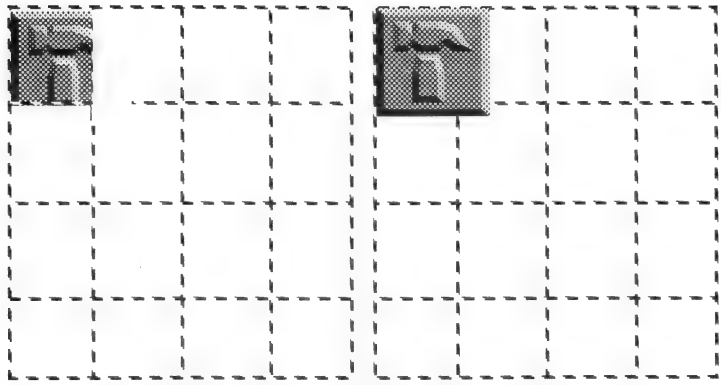


Figure 17.A Clipped Object and an Overlayed Object

15.2.3. Sizing a Grid

You can get the size of any grid cell, and you can change the size of a grid in one of two ways:

- Increase or decrease the *number* of grid cells
- Increase or decrease the *size* of a cell or cells while the number or rows and columns remains constant

To determine which of these ways you will size a grid, you must set its sizing policy through `CGrid::SetSizingPolicy`, which takes a value of either `ADJUSTCellSize` or `ADJUSTCellNumber`:

```
virtual void SetSizingPolicy(POLICY thePolicy);
```

Through `CGrid:: AdjustCells`, you can either maximize or minimize the size of a grid by finding the largest or the smallest object contained in the grid and making all grid cells that size.

15.3. Fixed and Variable Grids

Within a `CFixedGrid` object, all of the rows are the same size and all the columns are the same size. Thus, the cells of a fixed grid all have the same dimensions. When you change the dimensions of one cell in a fixed grid, then the size of all other cells in the grid also changes to match the new size. Fixed grids are useful when you want all the items shown in a grid to have equal weight, as in a panel of icons or in a list box. See Figure 18 for an example.

On the other hand, within a `CVariableGrid` object, the rows and columns have variable widths and heights, as in spreadsheets. Thus,

you can set the size of a row or column individually. There is a default width and height for all of the rows and columns, and unless you set the size of a specific row or column, it takes the default size. When you resize an entire variable grid, the default size changes, but any fixed sizes that have been set for specific rows and columns do not change. Figure 19 shows an example of a variable grid.

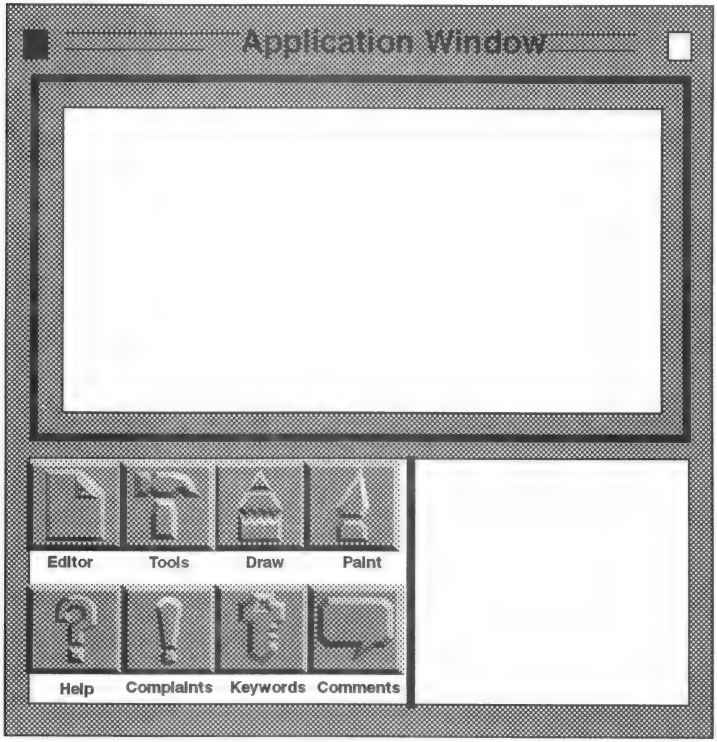


Figure 18. Use of a Fixed Grid, An Icon Panel

Order No.	Item Name	Quantity	Distribution Medium	Price	Total

Figure 19. Use of a Variable Grid, An Order Form

16

UTILITIES AND DATA STRUCTURES

16.1. Introduction

CMem	CGlue	CStartup
CError	CDesktop	CGlobalClassLib
	CEnvironment	CGlobalUser
	CGlobal	CResourceMgr
	CSwitchBoard	CPrintMgr

All of the classes in XVT-Power++'s application framework derive from one common class, `CBoss`, and they share a number of features: global data, message passing channels, data propagation, and so on. XVT-Power++ contains another set of classes that are independent of the application framework but widely used within it. These are the utility classes and data structures. The utility classes serves as a link between different XVT Portability Toolkit features and XVT-Power++. XVT-Power++ in no way claims to have a variety of data structures or a full set of container classes. In the future, XVT-Power++ may provide a much richer and more robust set of data structure classes, but so far this has not been the emphasis of XVT-Power++. Nonetheless, the classes XVT-Power++ does provide are very useful, and this chapter surveys the functionality that is available to you.

16.2. Managing Global Information

Through `CBoss`, all classes in XVT-Power++'s application framework have access to global XVT-Power++ information and to global user information, that is, application-specific information.

Two classes manage this global information: `CGlobalClassLib` and `CGlobalUser`. Each of these classes is instantiated once in an application. They are accessible through two pointers in the `CBoss` class: `G*` for `CGlobalClassLib` and `GU*` for `CGlobalUser`.

`CGlobalClassLib` provides an object that is automatically instantiated in XVT-Power++. It cannot be created or instantiated in any other way. It gives you access to `CDesktop`, another class discussed in this chapter, as well as to the print manager. It has pointers to the `CApplication` object and to the `CTaskWin` object. It supplies information to global flags, for example, about whether the application is terminating. Sometimes it is useful to know whether a destructor for a window is being called because the application is in an exit mode or simply because the window is being closed by a user. `CGlobalClassLib` also has a flag for text editing boxes, `itIsTextEvent` and a global ID count. A method called `GetID` gives you a unique ID number each time it is called and is used to provide ID numbers for different objects. Basically, `CGlobalClassLib` exists to supply information to different classes in the application framework.

Similarly, any global information that you wish to provide for your particular application is made available through `CGlobalUser`, which is initially empty. You must make a copy of the file containing the `CGlobalUser` definition and put in the application-specific data and functions. Then you must link the new file into the target application before the XVT-Power++ library, thus ensuring that your definition of `CGlobalUser` is used rather than XVT-Power++'s empty definition. For details on how to set the appropriate pointers and initialize your global information, see the sections on `CGlobalClassLib`, `CGlobalUser`, and `CBoss` in the *XVT-Power++ Reference*.

Another class that pertains to the global operation of XVT-Power++ is `CGlobal`, which is actually a file that contains definitions used by XVT-Power++. You should not modify this file, but you may want to refer to it occasionally to find out how something is defined. For example, you may need to know what resources XVT-Power++ defines internally or what the XVT-Power++ ID number base is. When you want to take a look at the file, see the section on `CGlobal` in the *XVT-Power++ Reference*.

16.3. Managing Resources

The class for managing resources is `CResourceMgr`, which is actually a file that is used specifically for defining icons used on X

platforms. Under X, icons are created somewhat differently than they are for other platforms, where they can simply be incorporated through a URL definition. For step-by-step information on how to build an icon or cursor resource, see the section on `CResourceMgr` in the *XVT-Power++ Reference*.

16.4. Setting Up the Environment

One data structure class that is pervasively used by XVT-Power++'s view classes, expect for the native views, is `CEnvironment`. This class contains different kinds of information about the environment: colors, types of pens and brushes, patterns, fonts, drawing modes, and so on. `CEnvironment` allows you to work in color or monochrome mode and to specify the way colors are used in monochrome mode, such as black for the foreground and white for the background, and so on. At any point, you can set the drawing environment before you do any drawing by calling the XVT Portability Toolkit drawing functions. For further information on setting the environment, see Section 6.4.2, entitled "Owners and Helpers" as well as the section on `CEnvironment` in the *XVT-Power++ Reference*.

16.5. Managing Window Layout Through The Desktop

Each time a window or a dialog is created in an XVT-Power++ application, the `CDesktop` object is notified. `CDesktop` is automatically instantiated within XVT-Power++. Of course, you can derive your own desktop, create it, and set it by informing `CGlobalClassLib` of the user-created desktop. Through `CDesktop`, you can define a window layout and keep track of the windows in your application. For example, you can find out how many windows are up, get a list of all open windows, and set a given window (that has been derived from `CModalWindow`) to be modal at any point. You can also find out which window is at the front of the window stack or notify the desktop to put a certain window at the front. For details on how the desktop works, see the section on `CDesktop` in the *XVT-Power++ Reference*.

16.6. Handling XVT Portability Toolkit Events

Another utility class that XVT-Power++ instantiates automatically is `CSwitchBoard`, one of the most heavily used classes. It provides an interface between XVT-Power++ and XVT Portability Toolkit events. `CSwitchBoard` has event handlers for dialogs, task windows,

windows, and so on. It is in charge of channeling events to the appropriate object. You do not have to initialize the switchboard, and you should not modify it. If you want to respond to a mouse event without notifying the switchboard, for example, you would override methods in `CWindow` rather than modify the switchboard.

16.7. Linking With XVT Portability Toolkit's Memory Management Functions

XVT-Power++ has a utility file named `CMem` that includes several kinds of definitions for `new` and `delete`, which are used by C++ in general and XVT-Power++ applications in particular. The definitions provide a link to some XVT Portability Toolkit memory management functions: `malloc`, `free`, and so on. Anytime you use `new` and `delete`, you are using an XVT Portability Toolkit memory allocation scheme for a particular platform. `CMem` also includes some memory debugging functions that are not supported. Sometimes they are used internally, but they can give you a list of objects that have been created and not deleted and so on.

16.8. Checking For Errors

XVT-Power++'s `CError` class is actually a link for a macro called `PWRAssert` that is used throughout XVT-Power++. It allows you to assert that certain things are true or false and to get an XVT Portability Toolkit error if something is wrong.

See Also: For a description of `PWRAssert`, see the section on `CError` in the *XVT-Power++ Reference.*

16.9. Printing

XVT-Power++'s interface to the XVT Portability Toolkit's printing facilities is called `CPrintMgr`. This class is in charge of queuing up data and printing it. Normally, when you want to print a view, you can call `DoPrint` inside the view. The actual implementation of printing is handled inside `CPrintMgr`.

16.10. Data Structures

While XVT-Power++ is not a data structures library, it does contain some data structures that it uses internally and that are available for your own use.

16.10.1. Lists

XVT-Power++ contains three different list classes. First, `CList` assumes no order for insertion of items and has an accompanying iterator class, `CIterator`. Next, `COrderedList` and `COrderedIterator` allow you to set up an ordered list and iterate over it. Finally, `CStringList` comprises an ordered list of strings and provides extra functionality for handling the string items. The main drawback of `CList` and `COrderedList` is that they use void pointers for inserting objects into a list. They do not embody the concept of using templates for inserting objects of different types. Thus, you must use casting when you insert items into or remove items from a list. All XVT-Power++ platforms will provide templates when they are available. There is also a `CSparseArray` class, which is similar to the ordered lists, except that it is a two-dimensional ordered list. It is an array that conserves memory by providing only enough space for the objects it contains.

16.10.2. Strings

XVT-Power++'s `CString` provides string functionality, including C++ streams. Reference counting has been implemented for both lists and strings, so they can be copied and passed around very efficiently.

16.10.3. The Coordinate System: CPoint, CRect, and CUnits

XVT-Power++'s `CPoint` class allows you to manage a coordinate system point in different possible types of units that you choose

through the `CUnits` class. Similarly, `CRect` allows you to manage a rectangular set of coordinates in different possible types of units.

See Also: For a full discussion of these classes, see Section 6.5 in this *Guide*.

17

LOGICAL UNITS

17.1. Introduction

CUnits

XVT-Power++'s `CUnits` class and the classes derived from it provide objects that allow the programmer to code viewable locations or regions using logical coordinates. These logical coordinates can be of different types: inches, centimeters, pixels, characters, or user-defined coordinates. When objects are drawn, these logical coordinates must be converted to a physical output device. The physical coordinates are always either dots on a printer or pixels on the screen that will be turned on or off according to the unit provided.

The most general class of the units classes is `CUnits`, which provides a generic unit that can be set to any desired type of logical to physical mapping. Each class derived from `CUnits` provides a specific type of mapping. `CCharacterUnits` maps logical units to physical units according to a base font while maintaining a proportion of character width and height in that font. `CInchUnits` maps units that represent inches on the screen to numbers of pixels. `CCentimeterUnits` maps units that represent centimeters on the screen to numbers of pixels.

When a `CUnits` object is created, the constructor requires you to pass in information about the horizontal and vertical screen mappings as well as the horizontal and vertical printer mapping:

```
CUnits(float theHScreenMapping = 1.0,
    float theVScreenMapping = 1.0,
    float theHPrinterMapping = 1.0,
    float theVPrinterMapping = 1.0,
    OutputDevice theDevice = SCREEN);
```

```
CUnits(float theHScreenMapping = 1.0,
    float theVScreenMapping = 1.0,
    float theHPrinterMapping = 1.0,
    float theVPrinterMapping = 1.0,
    OutputDevice theDevice = SCREEN);
```

If the horizontal screen mapping is 2, this means that there is a 2-to-1 ratio; that is, each horizontal unit will be translated into 2 pixels. The basic formula for calculating the physical coordinates is to take the logical coordinate and multiply it by the mapping: *physical = logical x mapping*. If there is one logical unit and a mapping of two, then the physical mapping is:

```
1 x 2 = 2
```

In addition to the screen and printer mappings, the `CUnits` constructor has a parameter for the output device to which you are currently translating, either the screen or a printer. At any time, you can change the output device that is being used for a particular `CUnits` object. Methods on the `CUnits` class allow you to set and get the output device, as well as the different screen and printer mappings. In addition, `CUnits` provides methods for converting *logical units to physical units* for both horizontal and vertical mappings. You can convert a `CPoint` or a `CRect` specified in units to an XVT Portability Toolkit `PNT` or `RCT`. Similarly, `CUnits` contains methods for converting *physical units to logical units*.

Note: These conversions are handled automatically inside XVT-Power++. Thus, while it is important to understand the logic of how units work, you will almost never need to call any of the `CUnits` mapping functions. All you must do is create a `CUnits` object and assign it to an object in the object hierarchy of the application framework. Everything else takes care of itself.

17.2. Dynamic Mapping

When a program is ported from one platform to another, it may suffer in appearance because different machines have different screen widths, heights, and resolutions. `CUnits` has a dynamic mapping capability that is designed to take care of this problem. An application is assumed to be developed on one platform and then ported to other platforms.

To specify the development platform, `CUnits::SetDevelopmentMetrics` is called with the metrics of the platform that was used for development: the device width, the device height, and the horizontal and vertical resolution. Once you have defined the development metrics, you can turn on dynamic

mapping via SetDynamicMapping. When the application executes, it compares the development metrics to the execution metrics. Thus, at run time, the application can compute the resolution, width, and height of the display device it is using. If the execution metrics are identical to the development metrics, then the program is running on a machine that is identical to the development platform, and no further computations are necessary. However, if the execution metrics differ from the development metrics, CUnits calculates the change of proportions in the resolution, width, and height; then it adjusts the horizontal and vertical mapping so that the application looks right on the new machine with its new metrics.

To make use of the dynamic mapping capability of CUnits, all you must do is call SetDevelopmentMetrics with the hard-coded values of the metrics of the machine you used for development and then turn on SetDynamicMapping. Everything else is automatic.

17.3. Owners of Units

Each CUnits object has a pointer to a boss object, and the pointer is the unit's *owner*. This owner is always the highest object in the object hierarchy that has been assigned the particular CUnits object. For example, when a CUnits object is assigned to a document, then every window and every view inside that document shares this CUnits object. The boss of the CUnits object is always the document. Although every window and every view is using the CUnits object. the document is the object that actually created it and therefore owns it. Whenever the units change, the CUnits object notifies its owner, and a message about the change, called UpdateUnits, trickles down to all objects that are sharing the units so they can accommodate the change. For example, if the units change for a window, it may need to resize itself, and every view inside it will also have to redraw itself according to new logical units. These updates occur automatically any time the units change. All updates occur through the UpdateUnits method. Whenever the units change, a set of UpdateUnits methods is called for all objects sharing the units so that they can update themselves.

In addition, when units are created (normally, on the heap) and assigned to an object by calling SetUnits, the ownership of the units belongs to the object to which you passed the units. The owner is responsible for deleting its CUnits object when it is itself deleted and for notifying any object sharing the CUnits object that the units are deleted. All of this is done internally, and you need not be concerned about it.

CUnits objects can be shared only in the manner described through the application framework. In other words, a subview can share the units of its enclosure, its window, its document, or its application. However, two different documents *cannot* share the same CUnits object as owners. If you want two documents to have the same kind of units, you must create two separate CUnits objects and set them for each of the documents. Documents can share the same CUnits object only through the CApplication object; that is, they can both use the CUnits owned by the application. In short, sharing can only occur downwards through the parent or child and cannot occur between peers.

17.4. Incorporating Units into XVT-Power++ Applications

By themselves, CUnits objects are simply data structures that do mappings between logical and physical coordinates. Their usefulness lies in the way that they are tied in to the rest of the application framework. Any object derived from CBoss can have its own units. The units hierarchy works in a very similar way to the environment hierarchy. For example, you can set the application's units using a method belonging to all objects derived from CBoss named SetUnits. Once you set the units object:

```
application -> SetUnits units object
```

the application owns those units, and any object in the application that does not have its own units set uses these units.

Suppose a view is nested three levels down inside a window. Before this view draws itself, it will check to see whether it has a units object of its own and whether it must do any units conversions. Normally, it does have its own units object and uses it. If it does not have its own units object, it checks to see whether its enclosure has a units object, and if it does, the view uses the units object of its enclosure. If the enclosure does not have a units object, then the view checks to see whether its enclosure's enclosure has one and so on all the way up to the window. From the window, the search continues to the document associated with the window, and finally, if the document does not own a units object, it checks the application object. The application may also not have its units set, in which case the whole application is simply using physical coordinates and no units are set.

Here we have described simply the abstraction of how units work. In reality, units are directly available to any object, and there is no

run-time search up the hierarchy because it would be too time-consuming. However, the semantics of how it all works are as described here.

A

ERROR MESSAGES

A.1. Introduction

This appendix lists the XVT-Power++ error messages according to the classes that generate them. The classes are presented in alphabetical order, and the error messages pertaining to each class are listed numerically according to the numbers XVT-Power++ assigns them. You can assume that the file containing the error messages for a class is the source (**.cxx**) file unless we explicitly state otherwise.

See Also: For further information about any of the classes and methods, consult the *XVT-Power++ Reference*.

There may be a disparity between the wording of the error message that appears on your screen and the wording of the same message as stated here; however, the error numbers match. In cases where there are disparities in the wording, you should assume that error message as stated in this appendix is correct.

A.2. Error Messages Listed by Class

A.2.1. CApplication

0, "Instantiating CApplication twice"

There can be only one instance of a `CApplication` object per application. The main `CApplication` constructor sets up the application object and acts as a safeguard, ensuring that only one application object is instantiated per application. The application should be created inside of `main`. This constructor simply sets up some global objects and some initial data structures—as should any derived class constructors. The constructor should take care of *only* these tasks. No other object

from the XVT-Power++ application framework should be created until the application receives a StartUp event.

1, "Instantiating CApplication twice"
There can be only one instance of a CApplication object per application. The CApplication copy constructor guarantees that no application object is instantiated twice.

2, "GU has not been deleted"
The user is responsible for deleting the CGlobalUser object. While the destructor for the application cleans up any global objects created by XVT-Power++, it is the responsibility of the derived constructor to delete any objects created by the derived application. If a global user object is created, be sure to delete it and set the GU pointer to NULL as shown here:

```
CMyApplication::~CMyApplication{void}
{

delete GU;

//Delete leaves GU undefined; set to NULL before
 terminating
GU=NULL;

. . .
. . .

}
```

3, "Adding Null Document"
CApplication::AddDocument takes a pointer to a document and adds the document to the application. This pointer cannot be a NULL pointer.

4, "CApplication: SetEnvironment called with NULL env"
CApplication::SetEnvironment sets the environment for a view, using the new environment that is passed to it. By default, views share their enclosure's environment. However, as soon as you use SetEnvironment to give a view an environment of its own, the view uses that environment instead of the shared environment. The SetEnvironment method takes a constant and a reference to an environment, which cannot have a value of NULL.

A.2.2. CBoss

100, "CBoss initialized with NULL gClassLib"
CBoss has two public data members, which are pointers. Anything that inherits through the CBoss object has access to the global data through one of these pointers. Both of these

static pointers are initialized to NULL, so you must set these pointers to point to an actual object.

`static CGlobalClassLib *G;`	Pwr global objects
`static CGlobalUser *GU;`	user-specific global objects

A CBoss initializer takes a pointer to an object of type CGlobalClassLib that contains global utilities for XVT-Power++.

101,"CBoss initialized with NULL gUser"
CBoss has two public data members, which are pointers. Anything that inherits through the CBoss object has access to the global data through one of these pointers. Both of these static pointers are initialized to NULL, so you must set these pointers to point to an actual object.

`static CGlobalClassLib *G;`	XVT-Power++ global objects
`static CGlobalUser *GU;`	user-specific global objects

A CBoss initializer takes a pointer to an object of type CGlobalUser that contains global utilities for a user application.

102, "NULL units passed to SetUnits"
The pointer passed to CUnits::SetUnits cannot have a value of NULL.

A.2.3. CDeskTop

200, "Desktop initialized with NULL application"
The CDesktop constructor takes a pointer to the application to which the desktop belongs. This pointer cannot have a value of NULL. One CDesktop object is created per application by the CApplication object. All core XVT-Power++ classes have access to the desktop through the global reference stored by CBoss::GetDesktop.

201, "Illegal cloning of the Desktop"
The CApplication object creates one CDesktop object per application. Copying/cloning of the desktop is not permitted.

202,"NULL window passed to SetFrontWindow"
CDeskTop::SetFrontWindow takes a pointer to a window and sets that window to the front of the window stack. This pointer cannot have a value of NULL.

203, "NULL window passed to SetFrontWindow"
`CDeskTop::SetFrontWindow` takes a pointer to a window and sets that window to the front of the window stack. This pointer cannot have a value of `NULL`.

204, "NULL window passed to FindWindow"
`CDeskTop::FindWindow`takes a pointer to a window and returns a Boolean value of `TRUE` if the window is in the desktop is in the desktop and `FALSE` if it is not. This pointer cannot have a value of `NULL`.

205, "NULL window passed to AddWindow"
`CDesktop::AddWindow` takes a pointer to a window and adds the window to the desktop. This pointer cannot have a value of `NULL`.

A.2.4. CDocument

300, "Adding NULL window to CDocument"
Given a window, `CDocument::AddWindow` adds a pointer to the document's list of windows. This pointer cannot have a value of `NULL`.

A.2.5. CGlobalClassLib

500, "NULL desktop passed to CGlobalClassLib"
`CGlobalClassLib::SetDesktop` takes a pointer to a desktop, which cannot have a value of `NULL`.

501, "NULL print manager passed to CGlobalClassLib"
The print manager passed to `CGlobalClassLib` cannot have a value of `NULL`.

A.2.6. CGlue

600, "Glue created with NULL owner"
The `CGlue` constructor takes a pointer to the view associated with the `CGlue` object. This pointer cannot have a value of `NULL`.

601, "Copying CGlue object is disabled"
The `CGlue` copy constructor disallows copying of the `CGlue` object.

A.2.7. CGrid

700, "NULL CView passed to Insert"
`CGrid` has two `Insert` methods. The one that is of concern here

inserts the designated view into the given row and column of the grid. It takes a pointer to the view that is to be inserted, a row number, and a column number. The coordinates of the view are adjusted so that it can be placed into the cell. The pointer to the view cannot have a `NULL` value.

701, "NULL CView passed to Insert"
`CGrid` has two `Insert` methods. The one that is of concern here takes a pointer to a view, examines the view's coordinates, and calculates the row and column of the grid into which it fits best. The pointer to the view cannot have a `NULL` value.

702, "NULL CView passed to PlaceView"
`CGrid::PlaceView` takes a pointer to a view and the row and column to which the view belongs and ensures that this view has been placed correctly within the cell. This method is called internally when a view is inserted or replaced. The pointer to the view cannot have a value of `NULL`.

703, "NULL view passed to ResetCell"
When a view is moved to a new cell within a grid, `CGrid::ResetCell` resets the cell to ensure it has the coordinates of its new location. This method takes a pointer to a view, which cannot have a value of `NULL`.

704, "NULL parameter(s) passed to GetPosition"
`CGrid` has two `GetPosition` methods. One of them gets the position of a given view; the other gets the position of a certain region. The one of concern here takes pointers to a view, a row, and a column. None of these pointers can have a value of `NULL`.

705, "NULL parameter(s) passed to GetPosition"
`CGrid` has two `GetPosition` methods. One of them gets the position of a given view; the other gets the position of a certain region. The one of concern here takes pointers to a `CRect`, a row, and a column. None of these pointers can have a value of `NULL`.

706 "NULL parameters passed to GetPosition"
`CGrid::GetPosition` takes pointers to a view, a row, and a column. None of these parameters can have a value of `NULL`.

A.2.8. CListBox

800, "ListBox item removal failed"
Given the number of a position on the list, `CListBox::RemoveLine` removes the string item that occupies this position. `thePosition` starts at zero (0) with the topmost item in the list. If there is no string item at the specified position, this error message is generated.

A.2.9. CListInline (CListInline.h)

900, "InsertInternal expects non-NULL previous link"

A `CList` object is composed of links, which are data structures that function as containers for individual items in a list. The `CLink` data structure contains three pointers: to the previous link, to the next link, and to the item of data in the link. The pointer to the previous link cannot have a value of `NULL`.

A.2.10. CList (CLists.cxx)

1000, "Inserting NULL item into COrderedList"

`CList::Insert` takes a pointer to an item and inserts the item into a list. The item is a void pointer to a generic object. This pointer cannot have a value of `NULL`.

1001, "Catenating NULL List"

The `CList::operator+=` concatenates two lists: a list with the current list. It takes a reference to a list that cannot have a value of `NULL`.

A.2.11. CNativeTextEdit

1100, "XVT cannot create text edit object"

This message is generated by the `CNativeTextEdit` constructor and typically means that there is insufficient memory to create the text edit object.

1101, "XVT cannot create text edit object"

This message is generated by the `CNativeTextEdit` copy constructor and typically means that there is insufficient memory to create the text edit object.

1102, "XVT cannot create text edit object"

This message is generated by the `CNativeTextEdit` assignment operator and typically means that there is insufficient memory to create the text edit object.

1103, "NULL buffer passed to GetLineInternal"

`CNativeTextEdit::GetLineInternal` returns one line of text. It takes the paragraph number and the line number and then appends the line to any text already contained in the given string buffer. The buffer cannot have a value of `NULL`.

1104, "NULL text passed to Append"

Given a text string, `CNativeTextEdit::Append` appends this string to whatever text is already inside a text edit object. This method returns a Boolean value of `TRUE` if the append operation succeeds. The text string passed cannot have a value of `NULL`.

1105, "NULL text passed to GetPartLine"

`CNativeTextEdit::GetPartLine` returns part of a line. This method takes the numbers of the starting and ending characters in the line, the paragraph number, the line number, and a string buffer. It appends the partial line mapped out by the first four parameters to any text contained in the string buffer. The string buffer cannot have a value of `NULL`.

1106, "NULL buffer passed to GetTextInternal"

`CNativeTextEdit::GetTextInternal` returns all of the text contained in a text box, performing the actual calculations within the text editing system. It takes the numbers of the first and last paragraphs, the numbers of the first and last lines, and the numbers of the first and last characters. Then it appends the text mapped out by these parameters to any text already contained in the given string buffer. The given string buffer cannot have a value of `NULL`.

1107, "NULL buffer passed to GetPartPar"

`CNativeTextEdit::GetPartPar` returns part of a paragraph. It has parameters for the starting character and ending character, the paragraph number, the starting line and ending line, and a string buffer. The partial paragraph mapped out by the first five parameters is appended to any text already contained in the string buffer. The string buffer cannot have a value of `NULL`.

1108, "NULL buffer passed to GetFullPar"

Given a paragraph number, `CNativeTextEdit::GetFullPar` returns the full paragraph of text, appending it to any text already contained in the given string buffer.

A.2.12. CNativeView

1200, "Control object cannot be multiply instantiated"

Each native view (control) can have only one instantiation at a time. For example, a scrollbar object can appear on the screen as only one scrollbar, not as two or three scrollbars. This message is generated by `CNativeView::CreateControl` when a user attempts to instantiate a control object more than once.

A.2.13. COrderedList

1300, "Inserting NULL item into COrderedList"

`COrderedList::Insert` inserts an item into a list. The item is a void pointer to a generic object. In addition this method takes a position in the list in which you want to insert the item. When you iterate over the list, the items will be put in order. You thus

do not have to worry about making the positions adjacent. When `Insert` puts an item into a position that already contains an item, it shifts the first item one position higher in the list. This method returns a Boolean value of `TRUE` if it has to do any shifting and `FALSE` if it does not. It generates this error message if the pointer to the item has a value of `NULL`.

1301, "Inserting NULL item into COrderedList"
`CorderedList::Replace` inserts an item into a position in an ordered list. If an item already occupies the given position, this method replaces the original item and returns it as a pointer. The void pointer is set to `NULL` if no item was replaced, and it points to an object if the item was replaced. The item parameter cannot be passed in to `Replace` with a value of `NULL`.

A.2.14. COval

1400, "Drawing COval with illegal coordinates"
This message is generated when the top-left corner and bottom-right corner of the `CRect` passed to `COval::Draw` have become interchanged.

A.2.15. CPoint

1500, "NULL local view passed to Globalize"
`CPoint::Globalize` converts the `CPoint` object's coordinates to global, window-relative coordinates. It takes a pointer to the local view from which the globalizing is done. This pointer cannot have a value of `NULL`.

1501, "NULL global view passed to Localize"
`CPoint::Localize` assumes that the `CPoint` is in global, window-relative coordinates and localizes the `CPoint` to the given view. The pointer to the given view cannot have a value of `NULL`.

A.2.16. CPolygon

1600, "Polygon created with NULL array of points"
`CPolygon::CreatePoints`, a method called by the `CPolygon` constructor, generates this message when there is an attempt to instantiate a polygon with a `NULL` array of points.

1601, "Polygon created with NULL list of points"
`CPolygon::CreatePoints`, a method called by the `CPolygon` constructor, generates this message when there is an attempt to instantiate a polygon with a `NULL` list of points.

A.2.17. CRect

1700, "NULL local view passed to Globalize"
`CRect::Globalize` converts the `CRect` object's coordinates to global, window-relative coordinates. It takes a pointer to the view from which the globalizing is done. This pointer cannot have a value of `NULL`.

1701, "NULL global view passed to Localize"
`CRect::Localize` assumes that the `CRect` is in global, window-relative coordinates and localizes the `CRect` to the given view. The pointer to the view cannot have a value of `NULL`.

A.2.18. CScroller

1800, "SetHIncrements called for for Scroller with no horizontal scrollbar"
`CScroller::SetHIncrements` sets the horizontal increments of the scroller, in pixels, for both line increments and page increments. It generates this message if it is called for a scroller with no horizontal scrollbar.

1801, "SetVIncrements called for Scroller with no vertical scrollbar"
`CScroller::SetVIncrements` sets the vertical increments of the scroller, in pixels, for both line increments and page increments. It generates this message if it is called for a scroller with no vertical scrollbar.

1802, "GetHLineIncrement called for Scroller with no horizontal scrollbar"
`CScroller::GetHLineIncrement` returns the number of pixels that have been set for a horizontal line increment. It generates this message if it is called for a scroller with no horizontal scrollbar.

1803, "GetHPageIncrement called for Scroller with no horizontal scrollbar"
`CScroller::GetHPageIncrement` returns the number of pixels that have been set for a horizontal page increment. It generates this message if it is called for a scroller with no horizontal scrollbar.

1804, "GetVLineIncrement called for Scroller with no vertical scrollbar"
`CScroller::GetVLineIncrement` returns the number of pixels that have been set for a vertical line increment. It generates this message if it is called for a scroller with no vertical scrollbar.

1805, "GetVPageIncrement called for Scroller with no vertical scrollbar"
`CScroller::GetVPageIncrement` returns the number of pixels that have been set for a vertical page increment. It generates this message if it is called for a scroller with no vertical scrollbar.

A.2.19. CSketchPad

1900, "GetSketchedRegion can be called only during a DoCommand"
`CSketchPad::GetSketchedRegion` returns a region—that is, a `CRect`—in coordinates that are relative to the sketchpad. These coordinates indicate the size of the region that has been dragged out. If you drag out a rectangular region, the rectangle is returned. If you drag out a line, a rectangular region that fits the whole line is returned. This method cannot be called directly.

1901, "GetStartPoint can be called only during a DoCommand"
`CSketchPad::GetStartPoint` returns the coordinate for the point at which mouse dragging started. This coordinate is relative to the sketchpad. This method cannot be called directly.

1902, "GetEndPoint can be called only during a DoCommand"
`CSketchPad::GetEndPoint` returns the coordinate for the point at which mouse dragging ended. This coordinate is relative to the sketchpad. This method cannot be called directly.

A.2.20. CSparseArray

2000, "Cloning of CSparseArray structures not yet implemented"
The `CSparseArray` assignment operator disallows copying of a sparse array.

2001, "Cloning of CSparseArray structures not yet implemented"
The `CSparseArray` copy constructor disallows copying of a sparse array.

2002, "Cloning of CSparseArrayIterators not yet implemented"
The `CSparseArrayIterator` copy constructor disallows copying of a sparse array iterator.

2003, "Cloning of CSparseArrayIterators not yet implemented"
The `CSparseArrayIterator` assignment operator disallows copying of a sparse array iterator.

2004, "NULL array passed to CSparseRowIterator"
The `CSparseRowIterator` constructor takes an array over

which to iterate and the number of the row through which it is to iterate. The pointer to the array cannot have a value of NULL.

2005, "Cloning CSparseRowIterator is not yet implemented"
The CSparseRowIterator copy constructor disallows copying of a CSparseRowIterator object.

2006, "Cloning CSparseRowIterator is not yet implemented"
The CSparseRowIterator assignment operator disallows copying of a CSparseRowIterator object.

2007, "NULL array passed to CSparseColIterator"
The CSparseColIterator constructor takes an array over which to iterate and the number of the column through which it is to iterate. The pointer to the array cannot have a value of NULL.

2008, "Cloning CSparseColIterator is not yet implemented"
The CSparseColIterator copy constructor disallows copying of a CSparseColIterator object.

A.2.21. CString

2100, "Resource ID passed to CString does not exist"
One of the CString constructors takes a resource ID number, which is the ID of a string resource that is defined in the URL file. It also takes a string size (in number of characters) for that resource ID. This message is generated if an ID for a nonexistent resource is passed in to the constructor.

2101, "NULL String object passed to operator+="
One of the CString::operator+= methods defines the plus-equal operator for concatenating strings. It takes a string to concatenate, which cannot have a value of NULL.

2102, "NULL String object passed to operator+="
One of the CString::operator+= methods defines the plus-equal operator for concatenating char pointers. The char pointer cannot have a value of NULL.

2103, "NULL initializer for CStringBody"
This message is generated when there is an attempt to construct a string with a NULL char pointer.

A.2.22. CSubview

2200, "NULL subview passed to AddSubview"
CSubview::AddSubview adds a pointer to a view to the view's list of views. This pointer cannot have a value of NULL.

2201, "NULL list passed to FindSubviews"
`CSubview::FindSubviews` method returns a list of every subview, nested or overlapping, that contains the given global, window-relative coordinate. This method takes not only a coordinate but also a list of views to search. This list cannot have a value of `NULL`.

2202, "View not nested passed to PlaceTopSubview"
In the case of overlapping views, `CSubview::PlaceTopSubview` places the given view at the top of the stack. It takes a pointer to a view, which cannot have a value of `NULL`.

2203, "View not nested passed to PlaceBottomSubview"
In the case of overlapping views, `CSubview::PlaceBottomSubview` places the given view at the bottom of the stack. It takes a pointer to a view, which cannot have a value of `NULL`.

A.2.23. CSwitchBoard

2300, "Instantiating CSwitchBoard twice"
There can be only one instantiation of `CSwitchBoard` per application. This message is generated by the `CSwitchBoard` constructor.

2301, "Instantiating CSwitchBoard twice"
There can be only one instantiation of `CSwitchBoard` per application. This message is generated by the `CSwitchBoard` copy constructor, whose purpose is to assure that a second `CSwitchBoard` object is not instantiated.

A.2.24. CUnits

2400, "SetBaseWindow called with NULL CWindow"
`CUnits::SetBaseWindow` takes a pointer to a CWindow object, which cannot have a value of `NULL`.

2401, "SetBaseFont failed to obtain printer mappings"
`CUnits::SetBaseFont` takes a pointer to a window, which cannot have a value of `NULL`.

A.2.25. CView

2500 "NULL enclosure passed to SetEnclosure"
`CView::SetEnclosure` sets the enclosure for the view to a different enclosure. It takes a pointer to a subview, which cannot have a value of `NULL`. Note that if a view belongs to a certain enclosure and then it is placed inside of a different

enclosure, its location relative to the window is also going to change.

A.2.26. CWindow

2600, "NULL CDocument passed to CWindow constructor"
One of the `CWindow` constructors takes a pointer to the window's document, as well as coordinates for the window, XVT Portability Toolkit-provided attributes for the window, a window type, and a menubar ID. The pointer to the document cannot have a value of `NULL`.

2601, "Creation of XVT window failed"

2602, "NULL CDocument passed to CWindow constructor"
One of the `CWindow` constructors takes a pointer to the window's document and an XVT Portability Toolkit window. The `CDocument` pointer cannot have a value of `NULL`.

2603, "NULL window passed to CWindow constructor"
One of the `CWindow` constructors takes a pointer to the window's document and an XVT Portability Toolkit window. The parameter for the XVT Portability Toolkit window cannot have a value of `NULL`.

2604, "CWindow copy constructor not implemented"
The `CWindow` copy constructor currently disallows copying of a window.

2605, "CWindow assignment operator not implemented"
The `CWindow` assignment operator currently disallows copying of a window.

2606, "Deleting an open CWindow"
The `CWindow` constructor cleans up the memory of the `CWindow` object. Before deleting the window, you must close it with a call to `CWindow::Close`.

2607, "CWindow cannot have a subview enclosure"
`CWindow::SetEnclosure` allows you to move a subview object from one enclosure to another. However, windows are special. `CWindow::SetEnclosure` has been disabled by overriding, which ensures that you cannot call `SetEnclosure` for a window because windows do not have an enclosure. On some platforms, a window might have a task window as an enclosure, but this is irrelevant to XVT-Power++.

2608, "Control ID with no matching control passed to DoControl"
`CWindow::ControlID` is called by the switchboard when it

receives a native view event. The window checks its list of native views for the ID that matches `theControlID`. If it finds the native view on its list, it passes a `DoHit` message directly to it. If it does not find a matching native view, it sends a message to its subviews to search for it. Usually, this method is called automatically, although you can call it if desired. This message is generated when a control ID for a nonexistent control is passed in.

A.2.27. CWireFrame

2700, "NULL enclosure passed to SetGroupEnclosure"
`CWireFrame::SetGroupEnclosure` sets the subview that will act as the enclosure of the group containing a wire frame. Only views sharing the same enclosure can be multiply selected into a group. Thus, two views selected in two separate windows are not part of the same group. `SetEnclosure` takes a pointer to a subview. This pointer cannot have a value of `NULL`.

A.2.28. CRadioGroup

2800, "RemoveButton called with invalid ID"
`CRadioGroup::RemoveButton` takes the ID number of a button and removes the button from the group. This message is generated when there is no button in the group with the ID number passed in.

A.2.29. NRadioButton

2800, "RemoveButton called with invalid ID"

A.2.30. NScrollText

3000, "VScroll called for window with no vertical scrollbar"
`NScrollText::VScroll` is called when a scrollbar receives a vertical scroll event. It scrolls the views contained inside the scroller. If there is no vertical scrollbar, then this message is generated.

3001, "VScroll called for window with no vertical scrollbar"
`NScrollText::VScroll` is called when a scrollbar receives a vertical scroll event. It scrolls the views contained inside the scroller. If there is no vertical scrollbar, then this message is generated.

3002, "ScrollTextCallback failed to find an NScrollText object"
`NScrollText::ScrollTextCallback` is called by the XVT

Portability Toolkit when the contents of the text box have been scrolled. It updates the scrollbars. One of its parameters identifies the specific text object that needs updating. This message is generated when `ScrollTextCallback` does not find the text object.

3003, "SetHIncrements called for non-horizontal scrolling text system"
`NScrollText::SetHIncrements` sets the horizontal increments of the scrolling text object, in pixels, for both line increments and page increments. It generates this message if it is called for a text edit object with no horizontal scrollbar.

3004, "SetVIncrements called for NScrollText object with no vertical scrollbar"
`NScrollText::SetVIncrements` sets the vertical increments of the scrolling text object, in pixels, for both line increments and page increments. It generates this message if it is called for a text edit object with no vertical scrollbar.

3005, "GetHLineIncrement called for NScrollText object with no horizontal scrollbar"
`NScrollText::GetHLineIncrement` returns the number of lines that have been set for a horizontal line increment during scrolling. It generates this message if it is called for a text edit object with no horizontal scrollbar.

3006, "GetHPageIncrement called for NScrollText with no horizontal scrollbar"
`NScrollText::GetHPageIncrement` returns the number of lines that have been set for a horizontal page increment during scrolling. It generates this message if it is called for a text edit object with no horizontal scrollbar.

3007, "GetVLineIncrement called for NScrollText with no vertical scrollbar"
`NScrollText::GetVLineIncrement` returns the number of lines that have been set for a vertical line increment during scrolling. It generates this message if it is called for a text edit object with no vertical scrollbar.

3008, "GetVPageIncrement called for NScrollText object with no vertical scrollbar"
`NScrollText::GetVPageIncrement` returns the number of pixels that have been set for a vertical page increment during scrolling.

A.2.31. NTextEdit

3100, "NULL text passed to AppendToParagraph"
`NTextEdit::AppendToParagraph` appends a paragraph to a text box. It takes a string buffer and a paragraph number. The paragraph number must be a valid, existing paragraph number, and the string buffer cannot have a value of `NULL`.

3101, "NULL text passed to SetParagraph"
`NTextEdit::SetParagraph` sets a paragraph. This method takes a string buffer and a paragraph number. The paragraph number must be a valid, existing paragraph number, and the string buffer cannot have a value of `NULL`.

3102, "SetParagraph: invalid paragraph number"
`NTextEdit::SetParagraph` sets a paragraph. This method takes a string buffer and a paragraph number. The paragraph number must be a valid, existing paragraph number, and the string buffer cannot have a value of `NULL`.

3103, "SetLine not yet implemented"
`NTextEdit::SetLine` is not yet implemented. When it is, it will set a line within a paragraph. This method will take a string buffer, a paragraph number, and a line number. Both the paragraph number and the line number must be valid, existing numbers.

A.2.32. NWinScrollBar

3200, "Calling disabled copy constructor"
The `NWinScrollBar` copy constructor disallows copying of an `NWinScrollBar` object. You cannot copy one of these scrollbars to another because you cannot instantiate them directly. They are part of a window, and the window creates them.

3201, "Calling disabled assignment operator"
The `NWinScrollBar` assignment operator disallows copying of an `NWinScrollBar` object. You cannot copy one of these scrollbars to another because you cannot instantiate them directly. They are part of a window, and the window creates them.

XVT-POWER++

INDEX

Symbols

A

B

C

D

E

F

G

M

N

S

T

U

V

W

X

Z